Science of Computer Programming 200 (2020) 102544

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Efficient counter-factual type error debugging

Sheng Chen *, Baijun Wu

CACS, UL Lafayette, United States of America

ARTICLE INFO

Article history:
Received 21 November 2019
Received in revised form 18 July 2020
Accepted 31 August 2020
Available online 10 September 2020

Keywords:
Efficient type-error debugging
Variational typing
Principal typing

ABSTRACT

Providing effective error messages in response to type errors continues to be a challenge in functional programming. Type error messages often point to bogus error locations or lack sufficient information for removing the type error, making error debugging ineffective. Counter-factual typing (CFT) addressed this problem by generating comprehensive error messages with each message includes a rich set of information. However, this comes with a cost of huge computations, making it too slow for interactive use. In particular, our recent study shows that programmers usually have to go through multiple iterations of updating and recompiling programs to remove a type error. Interestingly, our study also finds that program updates are minor in each iteration during type error debugging. We exploit this fact and develop *e*CFT, an efficient version of CFT, which doesn't recompute all error fixes from scratch for each updated program but only recomputes error fixes that are changed in response to the update. Our key observation is that minor program changes lead to minor error suggestion changes. *e*CFT is based on principal typing, a typing scheme more amenable to reuse previous typing results. We have evaluated our approach and found it is about 12.4× faster than CFT in updating error fixes.

1. Introduction

Type inference allows programs to be statically typed, even without the presence of type annotations. A well-known problem in type inference is that it is very hard to locate the real error cause and generate informative feedback once type inference fails. Practical compilers pay little attention to address this problem. They usually report the place that type inference first fails as the error cause and often report errors in their internal jargon. As a result, understanding type error messages is a main challenge in learning functional programming [33,3,48].

This problem has also been intensively studied over the last three decades from different directions. One direction aims to find the most likely error causes [29,34,18,27,24,57]. As an example, consider the following ill-typed expression, which we will use throughout the paper.

```
rank = λx.(x '1',x True)
```

Helium [26], a Haskell compiler integrated with the type error debugging method developed in [27], produces the following error message. GHC 8.0.2, the de facto Haskell compiler, generates a similar message that blames the same location.

* Corresponding author.
E-mail addresses: chen@louisiana.edu (S. Chen), bj.wu@louisiana.edu (B. Wu).

https://doi.org/10.1016/j.scico.2020.102544
0167-6423/

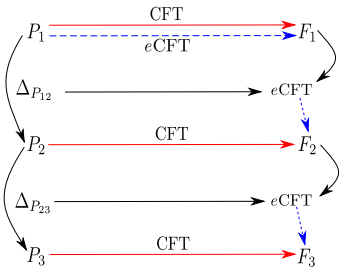

Fig. 1. Work flows of CFT and *e*CFT.

```
Type error in application
 expression       : x True
 function         : x
   type           : Char -> a
 1st argument     : True
   type           : Bool
   does not match : Char
```

This error message blames `True` as the error cause. While changing `True` may remove the type error, this is not the only possible fix. In fact, changing any of `x` (either occurrence), `'1'`, or `True` can remove the type error. We seem to lack enough context to justify that `True` is more likely the error source than other locations.

This example demonstrates the value of type error slicing [47,22,42], which returns all program locations that may contribute to the type error and excludes those don't. However, a problem with this approach is that the programmer still has to decide the real error cause among the returned slice, which could be comparable to the original program in size [27,7,10]. Recently, [39] improved this by finding all possible error causes and suggesting one location at a time.

Like error slicing, counter-factual typing (CFT) [7] also finds all possible error locations in the leaves and their combinations of the program AST. However, unlike error slicing, CFT also comes with a change suggestion for each identified location. This suggestion includes the type the identified location has in the original program, the type it ought to have to remove the type error, and the result type of the changed expression if the suggestion is applied. Since some locations are more likely to be the error source than others in most common cases, CFT ranks all fixes and presents them to programmers iteratively. The evaluation result showed that CFT achieved better precision than state-of-the-art approaches when considering the first suggestions and performed even better when considering also later suggestions [7].

One problem with CFT, however, is the long response time. To find all possible error locations and all change suggestions, CFT has to perform a lot of computations. Although CFT uses variational typing [14] to reuse typing results, it can still take dozens of seconds to deliver the first error message for programs within 100 LOC. This makes CFT too slow for interactive use. In particular, our recent study of mining a program database [23,50,25] shows that in average students take about 29 steps to fix a type error with a maximum of 359 steps. The details of the study are given in Appendix A.

Fortunately, an accompanying finding of the study is that, during error debugging, the change between two consecutive versions is very minor. In more than 80% cases, the change is within 10% of the old program. This result encourages us to compute error fixes incrementally rather than recompute all error fixes from scratch as programs are updated.

In this work, we develop *e*CFT, an efficient version of CFT. Fig. 1 depicts the difference between *e*CFT and CFT. We use blue, dashed arrows to denote computations of *e*CFT and red, solid arrows to denote those of CFT. P_i denotes the ith version of a program used in compilation, and F_i is the set of all error fixes produced by CFT (*e*CFT produces the same set of error fixes) for P_i. In addition, let $\Delta_{P_{ij}}$ denote the program difference between P_i and P_j. While CFT recomputes error fixes every time the program is updated, *e*CFT reuses earlier error fixes and program differences to more efficiently compute all error fixes. For example, CFT computes F_j by using P_j only, while *e*CFT takes F_i and $\Delta_{P_{ij}}$ as inputs to compute F_j.

In Summary, this paper makes the following contributions:

1. In Section 4, we present our first technical innovation of a declarative specification of typing applications, which simplifies the type system of *e*CFT. Earlier work that combines variational types and error types uses an operational rule to type applications, having to explicitly pass around and propagate error types. The type system in this paper averts this problem.
2. The CFT's typing rules pass around the type environment and its implementation is based on the algorithm $\mathcal{W}$, making CFT hard to incrementalize. To address this issue, we base *e*CFT on principal typing, a typing scheme more amenable to incremental updates. We present the typing rules of *e*CFT in Section 5. A subtle issue in the type system is about dealing with unbound variables, which we handle nicely with our previous contribution.
3. We present three type inference algorithms in Sections 6, 7, and 8. While the algorithm in Section 6 recomputes all error fixes as programs are updated, the algorithm in Section 7 only retypes subexpressions that are affected by the program updates, and the algorithm in Section 8 improves the second one by solving the variational unification problems incrementally, which is also our second technical innovation.

4. We extensively evaluate the performance of *e*CFT. The result shows that in more than 80% cases *e*CFT is 12.4× faster than CFT of computing error fixes in response to program updates.

This paper is an extended version of a conference paper published at TASE 2019 with the following additions and updates. (1) We improved the exposition of this paper with more explanations and examples. For example, we added Figs. 1, 3, 4, 8, and 9. We added examples throughout the paper. (2) We extended our formalization to support polymorphic let-expressions and conditionals. We extended the type system (Section 5) and type inference algorithms (Sections 6 and 8) correspondingly. (3) We extended our evaluation and observed an interesting phenomenon in the evaluation result and gave potential explanations for that (Section 9). (4) We included the statistics of an empirical study (Appendix A). about the number of iterations students needed to fix a type error and the change ratio between each two steps.

In the rest of the paper, we present the background of variational typing (Section 2) with a focus on the typing rule for function applications (Section 2.3), present the background of principal typing in Section 3, discuss related work in Section 10, and conclude the paper in Section 11.

2. Variational typing

As already mentioned in Section 1, CFT relies on variational typing [14,12] to compute informative error messages for all possible error locations. In fact, neither CFT nor *e*CFT is feasible without reusing typing information, the core idea of variational typing. This section presents variational typing.

Variational expressions are obtained by extending normal expressions (*plain* expressions) with named *choices* [21]. For example, the expression `e = succ` $A\langle$`1, 'a'`$\rangle$ contains a choice A, which has two alternatives, `1` and `'a'`. We will use d to range over choice names. While in general choices can be n-ary, binary choices are sufficient in this work, and we will consider them only.

An important notion in variation representations is *selector*s that have the form $d.i$, where d is a choice name and i is an alternative index. We call a set of selectors a decision and use δ to range over decisions. Choices can be eliminated through a process called *selection*, which takes in an expression e and a selector $d.i$ and replaces each occurrence of the choice d in e with its ith alternative. Selection extends naturally to decisions, by iteratively selecting with all of the selectors in the decision. We write $\lfloor e \rfloor_{d.i}$ and $\lfloor e \rfloor_{\delta}$ for selections. For example, $\lfloor$`succ` $A\langle$`1, 'a'`$\rangle\rfloor_{A.1}$ yields `succ 1`. Plain expressions can be obtained by eliminating all the choices in a variational expression. Choices with the same name are synchronized and those with different names are independent. Therefore, the variational expression $A\langle$`succ, odd`$\rangle$ $A\langle$`1, 'a'`$\rangle$ encodes two plain expressions: `succ 1` and `odd 'a'`, while $A\langle$`succ, odd`$\rangle$ $B\langle$`1, 'a'`$\rangle$ encodes four: `succ 1`, `succ 'a'`, `odd 1`, and `odd 'a'`.

Similarly, we have variational types. The notions and definitions of variational expressions carry over naturally to variational types. We will use τ and ϕ to range over plain types and variational types, respectively.

2.1. Selection and semantics of variational types

Since selection and semantics of variational types serve an important foundation to formally discuss our approach, we present them below. Both definition assume the type definition includes γ for ranging over constant types (such as `Int` and `Bool`), α for ranging over type variables, the variational type, and the function type.

$$\begin{aligned} \lfloor \gamma \rfloor_{d.i} &= \gamma & \lfloor d\langle \phi_1, \phi_2 \rangle \rfloor_{d.1} &= \lfloor \phi_1 \rfloor_{d.1} \\ \lfloor \alpha \rfloor_{d.i} &= \alpha & \lfloor d\langle \phi_1, \phi_2 \rangle \rfloor_{d.2} &= \lfloor \phi_2 \rfloor_{d.2} \\ \lfloor \phi_1 \to \phi_2 \rfloor_{d.i} &= \lfloor \phi_1 \rfloor_{d.i} \to \lfloor \phi_2 \rfloor_{d.i} & \lfloor d\langle \phi_1, \phi_2 \rangle \rfloor_{d_1.i} &= d\langle \lfloor \phi_1 \rfloor_{d_1.i}, \lfloor \phi_2 \rfloor_{d_1.i} \rangle \end{aligned}$$

Selecting γ or α with any selector yields themselves. Selecting a function type recursively selects its parameter and return types. Selecting a variational type $d\langle \phi_1, \phi_2 \rangle$ with a selector $d.1$ recursively selects the first alternative ϕ_1 with $d.1$. The reason for the recursive selection is that variations with the same name may be nested. For example, selecting $A\langle A\langle$`Int, Bool`$\rangle$, `Int`$\rangle$ with $A.1$ yields `Int`. Without recursive selection, the result would be $A\langle$`Int, Bool`$\rangle$, which is incorrect. The recursive selection addresses this issue. Similarly, selecting $d\langle \phi_1, \phi_2 \rangle$ with $d.2$ will recursively select the right alternative ϕ_2 with $d.2$. Selecting $d\langle \phi_1, \phi_2 \rangle$ with a selector whose variation name is not d will push the selection onto its alternatives, as specified by the last case of the definition of selection.

The semantics of a variational type is a mapping that maps decisions to plain types. Essentially, the semantics of a variational type specifies how plain types are encoding using variations. The semantics of γ (α) contains only one element, which maps the empty decision to γ (α), respectively.

$$[\![\gamma]\!] = \{(\{\}, \gamma)\}$$

$$[\![\alpha]\!] = \{(\{\}, \alpha)\}$$

$$[\![d\langle \phi_1, \phi_2 \rangle]\!] = \begin{cases} [\![\lfloor \phi_1 \rfloor_{d.1}]\!] & \lfloor \phi_1 \rfloor_{d.1} = \lfloor \phi_2 \rfloor_{d.2} \\ \{(\{d.1\} \cup \delta, \tau) | (\delta, \tau) \in [\![\lfloor \phi_1 \rfloor_{d.1}]\!]\} & \\ \cup \{(\{d.2\} \cup \delta, \tau) | (\delta, \tau) \in [\![\lfloor \phi_2 \rfloor_{d.2}]\!]\} & \text{otherwise} \end{cases}$$

$$[\![\phi_1 \to \phi_2]\!] = \{(\delta_1 \cup \delta_2, \tau_1 \to \tau_2) | (\delta_1, \tau_1) \in [\![\phi_1]\!] \wedge (\delta_2, \tau_2) \in [\![\phi_2]\!] \wedge \delta_1 \sim \delta_2\}$$

The semantics for $d\langle\phi_1, \phi_2\rangle$ needs to consider several cases. First, if ϕ_1 and ϕ_2 are the same, then essentially the variation d is redundant. Consequently, the semantics of $d\langle\phi_1, \phi_2\rangle$ is the same as that of ϕ_1 (or ϕ_2). In addition to accounting for this, the case (with a condition $\lfloor\phi_1\rfloor_{d.1} = \lfloor\phi_2\rfloor_{d.2}$) further handles variation nesting. For example, the semantics of $A\langle A\langle \texttt{Int}, \texttt{Bool}\rangle, \texttt{Int}\rangle$ should be the same as that of $A\langle \texttt{Int}, \texttt{Int}\rangle$ because $\lfloor A\langle A\langle \texttt{Int}, \texttt{Bool}\rangle, \texttt{Int}\rangle\rfloor_{A.1} = \lfloor A\langle \texttt{Int}, \texttt{Int}\rangle\rfloor_{A.1} = \texttt{Int}$ and $\lfloor A\langle A\langle \texttt{Int}, \texttt{Bool}\rangle, \texttt{Int}\rangle\rfloor_{A.2} = \lfloor A\langle \texttt{Int}, \texttt{Int}\rangle\rfloor_{A.2} = \texttt{Int}$. Second, if $\lfloor\phi_1\rfloor_{d.1}$ and $\lfloor\phi_2\rfloor_{d.2}$ are not the same, then the semantics of $d\langle\phi_1, \phi_2\rangle$ is the union of the semantics of ϕ_1 (but need to extend each decision with $d.1$) and of ϕ_2 (but need to extend each decision with $d.2$).

The semantics of the function type $\phi_1 \to \phi_2$ composes that of ϕ_1 and ϕ_2. The composition needs to ensure that the decisions δ_1 (for the semantics of ϕ_1) and δ_2 (for the semantics of ϕ_2) are consistent, expressed through the notation $\delta_1 \sim \delta_2$, which is defined as follows

$$\delta_1 \sim \delta_2 \stackrel{\text{def}}{=} \nexists d.d.1 \in \delta_1 \wedge d.2 \in \delta_2 \vee d.2 \in \delta_1 \wedge d.1 \in \delta_2$$

Essentially, $\delta_1 \sim \delta_2$ if there is no variation such that one of δ_1 and δ_2 contains $d.1$ and the other contains $d.2$ for some d. This condition is needed to prevent the creation of inconsistent decisions such as $\{A.1, A.2\}$. This decision is inconsistent because applying it to some type means to take both alternatives of the variation A, which is not a well-defined operation in the Choice Calculus [21].

2.2. Type equivalence

There are two different ways to type variational programs. A naive way is to individually type all the plain programs generated from the variational program. Note that the number of plain programs is exponential in the number of different named choices. Therefore, this method becomes impractical as it's common for a variational program to have thousands of independent choices [6].

A more scalable way is variational typing, which types variational programs once without generating plain programs. The central idea of variational typing is reuse. In our previous work [14], we identified three opportunities for reusing typing information in variational typing.

The conventional rule for typing a function application has the following form [40].

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_1 = \tau_2 \to \tau}{\Gamma \vdash e_1\ e_2 : \tau} \text{ T-App-Std}$$

Intuitively, for an application to be well typed, the function must have a function type (τ_1) and the parameter type (τ_2) of the function be the same as the type of the argument (τ_2). And if this is the case, the type of the whole application is the return type of the function (τ). The "=" sign in the third rule serves as an equality testing, which requires that its two sides be syntactically the same.

However, the equality testing becomes too restrictive for typing variational expressions. Consider, for example, typing the expression $\texttt{odd}\ A\langle 1, 2\rangle$. The function odd has the type $\texttt{Int} \to \texttt{Int}$, and the parameter type of odd is Int. The type of the argument $A\langle 1, 2\rangle$ is $A\langle \texttt{Int}, \texttt{Int}\rangle$. Since Int is not equal to ($\neq$) $A\langle \texttt{Int}, \texttt{Int}\rangle$, we can not use the application rule T-App-Std to type the expression $\texttt{odd}\ A\langle 1, 2\rangle$. Nevertheless, if we generate the plain expressions from $\texttt{odd}\ A\langle 1, 2\rangle$, we can check that both plain expressions (odd 1 and odd 2) are well typed. Thus, we expect that $\texttt{odd}\ A\langle 1, 2\rangle$ itself be well typed.

Our solution is relaxing the equality testing relation (=) in the typing rule to an equivalence relation ($\equiv$), yielding the following typing rule.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_1 \equiv \tau_2 \to \tau}{\Gamma \vdash e_1\ e_2 : \tau} \text{ T-App-Vari}$$

Two types ϕ_1 and ϕ_2 are equivalent, written as $\phi_1 \equiv \phi_2$, if ϕ_1 and ϕ_2 have the same semantics,[1] as defined in Section 2.1. Intuitively, $\phi_1 \equiv \phi_2$ if selecting ϕ_1 and ϕ_2 with the same decision always yield the same plain type. Fig. 2 gives the full set of type equivalence rules. The first three rules state that type equivalence is reflexive, symmetric, and transitive. Rule E4 says that a choice whose alternatives are identical is equivalent to its alternatives.

In rule E5, we use $\phi[\ ]$ to denote a type term with a hole where we can plug in a type. Rule E5 states that type equivalence is a congruence. Due to choice synchronization, if a choice type is nested in another choice type with the same name, then one alternative of the inner choice is unreachable. For example, in $A\langle A\langle \phi_1, \phi_2\rangle, \phi_3\rangle$, ϕ_2 is unreachable since selecting the type with $A.1$ gives us ϕ_1 and selecting it with $A.2$ yields ϕ_3. Rule E6 states that eliminating unreachable alternatives preserves type equivalence. Finally, rule E7 says that we may commute the function and choice type constructors. We defer a detailed discussion of these rules to [14] or [6].

Based on the rule E4, we know $\texttt{Int} \equiv A\langle \texttt{Int}, \texttt{Int}\rangle$. Combining this with the typing rule T-App-Vari, the expression $\texttt{odd}\ A\langle 1, 2\rangle$ has the type Int.

[1] In practice, however, deciding type equivalence is more efficiently realized through term rewriting, a technique developed in [14].

$$\text{E1}\ \ \phi \equiv \phi \qquad \text{E2}\ \ \frac{\phi \equiv \phi_1}{\phi_1 \equiv \phi} \qquad \text{E3}\ \ \frac{\phi \equiv \phi_1 \quad \phi_1 \equiv \phi_2}{\phi \equiv \phi_2} \qquad \text{E4}\ \ d\langle \phi, \phi \rangle \equiv \phi \qquad \text{E5}\ \ \frac{\phi_1 \equiv \phi_2}{\phi[\phi_1] \equiv \phi[\phi_2]}$$

$$\text{E6}\ \ d\langle \phi_1, \phi_2 \rangle \equiv d\langle \lfloor \phi_1 \rfloor_{d.1}, \lfloor \phi_2 \rfloor_{d.2} \rangle \qquad \text{E7}\ \ d\langle \phi_1, \phi_2 \rangle \to d\langle \phi_1', \phi_2' \rangle \equiv d\langle \phi_1 \to \phi_1', \phi_2 \to \phi_2' \rangle$$

Fig. 2. Variational type equivalence.

2.3. Error-tolerant variational typing

Variational typing assigns types to expressions that generate only well-typed plain expressions. For example, it fails to assign a type to the expression `odd` $A\langle$`1, True`$\rangle$ since the plain expression `odd True` is ill typed. In practice, it's very useful to assign types to its well-typed variants even a variational program contains type errors in other variants [12]. We addressed this problem by designing an error-tolerant type system, where type errors are represented explicitly by $\bot$ and variants that contain type errors receive this type [12]. For example, `odd` $A\langle$`1, True`$\rangle$ has the type $A\langle$`Bool`$, \bot\rangle$, indicating that `odd 1` has the type `Bool` and `odd True` is ill typed.

The typing rule for function applications is very complicated in the error-tolerant type system since applications can introduce type errors in many different ways. In particular, the rule has to propagate errors from both the function and argument types and generate errors when the parameter type fails to match the type of the argument exactly. We handle all these situations with the following single rule plus an additional device π called typing pattern.

$$\pi ::= \bot \mid \top \mid d\langle \pi, \pi \rangle$$

$$\frac{\Gamma \vdash e_1 : \phi_1 \qquad \Gamma \vdash e_2 : \phi_2 \qquad \phi_2' \to \phi' = \uparrow(\phi_1) \qquad \pi = \phi_2' \bowtie \phi_2 \qquad \phi = \pi \lhd \phi'}{\Gamma \vdash e_1\ e_2 : \phi}\ \text{T-App-Err}$$

A typing pattern π indicates which variants of an expression have type errors. It consists of $\bot$ for ill-typed variants, $\top$ for well-typed variants, and choice patterns for variational expressions.

The rule is quite complicated, relying on three operations: $\uparrow$, $\bowtie$, and $\lhd$. We will briefly discuss this rule, mainly for the purpose of comparing it with our new formalization in Section 4. Intuitively, the first two premises retrieve the types of the function (e_1) and the argument (e_2). The next three premises introduce and propagate errors, if there are any, to ϕ, which is the result type of the application ($e_1\ e_2$).

After retrieving the types for the function and the argument, the rule uses the operation $\uparrow(\phi)$ to transform ϕ into a function type and introduces $\bot$s when necessary. For example, $\uparrow($`Int`$) = \bot \to \bot$ and $\uparrow(A\langle$`Int` $\to$ `Int, Int` $\to$ `Bool`$\rangle) =$ `Int` $\to A\langle$`Int, Bool`$\rangle$.

The operation $\phi_2' \bowtie \phi_2$ determines how well ϕ_2' matches ϕ_2. It returns a typing pattern containing $\top$s for variants that ϕ_2' and ϕ_2 match and $\bot$s for those that they don't. Its formal definition is given below, reproduced from [12].

The first rule specifies that two same types match completely, indicated by the result $\top$. Matching variational types will push the matching onto their alternatives, as shown by the next three rules. Matching a type against a $\bot$ always fails.

$$\begin{aligned}
\phi \bowtie \phi &= \top \\
d\langle \phi_1, \phi_2 \rangle \bowtie d\langle \phi_1', \phi_2' \rangle &= d\langle \phi_1 \bowtie \phi_1', \phi_2 \bowtie \phi_2' \rangle \\
d\langle \phi_1, \phi_2 \rangle \bowtie \phi &= d\langle \phi_1 \bowtie \phi, \phi_2 \bowtie \phi \rangle \\
\phi \bowtie d\langle \phi_1, \phi_2 \rangle &= d\langle \phi \bowtie \phi_1, \phi \bowtie \phi_2 \rangle \\
\bot \bowtie \phi = \phi \bowtie \bot &= \bot \\
\phi_1 \to \phi_1' \bowtie \phi_2 \to \phi_2' &= \phi_1 \bowtie \phi_2 \otimes \phi_1' \bowtie \phi_2' \\
\phi \bowtie \phi' &= \bot \qquad \textit{(otherwise)}
\end{aligned}$$

When matching two function types, we first match their respective parameter and return types and then combine the matching results with the $\otimes$ operation. This operation can be understood as the logic "and" operation when view $\top$ and $\bot$ as truth values "True" and "False", respectively. Its formal definition is as follows.

$$\bot \otimes \pi = \bot \qquad \top \otimes \pi = \pi \qquad d\langle \pi_1, \pi_2 \rangle \otimes \pi = d\langle \pi_1 \otimes \pi, \pi_2 \otimes \pi \rangle$$

Finally, if none of the above case matches and the types are different, the matching yields a $\bot$. Based on this definition, `Int` $\bowtie$ `Int` $= \top$ (the first case of $\bowtie$), `Int` $\bowtie$ `Bool` $= \bot$ (the last case of $\bowtie$), `Int` $\bowtie B\langle$`Int, Bool`$\rangle = B\langle\top, \bot\rangle$ (the fourth case of $\bowtie$), and $A\langle$`Bool, Int`$\rangle \bowtie B\langle$`Int, Bool`$\rangle$ = $A\langle$`Bool` $\bowtie B\langle$`Int, Bool`$\rangle$, `Int` $\bowtie B\langle$`Int, Bool`$\rangle\rangle$ = $A\langle B\langle\bot, \top\rangle, B\langle\top, \bot\rangle\rangle$.

The operation $\pi \lhd \phi'$ replaces each occurrence of $\top$ in π with ϕ' and leaves $\bot$s untouched, meaning that once errors have occurred they can't disappear along the typing process. For example, $B\langle\top, \bot\rangle \lhd A\langle$`Int, Bool`$\rangle = B\langle A\langle$`Int, Bool`$\rangle, \bot\rangle$.

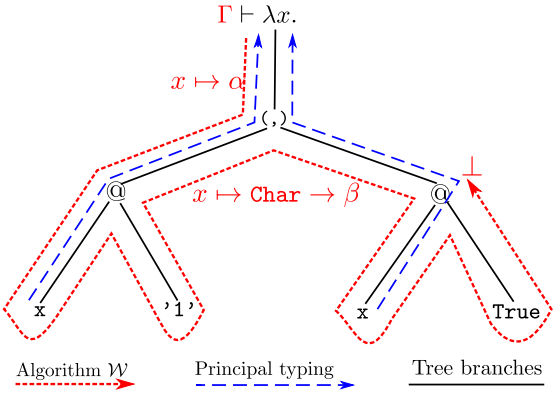

Fig. 3. The environment flows in different directions for algorithm $\mathcal{W}$ and principal typing.

Based on the above operations, we can compute the type of $A\langle \texttt{succ}, \texttt{odd}\rangle\ B\langle \texttt{1}, \texttt{True}\rangle$ as follows. First, $A\langle \texttt{succ}, \texttt{odd}\rangle$ has the type $A\langle \texttt{Int} \to \texttt{Int}, \texttt{Int} \to \texttt{Bool}\rangle$ and $B\langle \texttt{1}, \texttt{True}\rangle$ has the type $B\langle \texttt{Int}, \texttt{Bool}\rangle$. Next, by combining the last example from each of the previous three paragraphs, we get the result type $B\langle A\langle \texttt{Int}, \texttt{Bool}\rangle, \bot\rangle$.

In general, to make a type system error-tolerant, every type equality in the typing rules has to be replaced with these three operations. Both *e*CFT and CFT have to be error-tolerant since their main goal is to propose suggestions for ill-typed expressions. It's easy to see that these operations appear several times in CFT [7] and will appear more often in *e*CFT since it is based on principal typing, which has to equate all assumptions for the same variable quite often. Since these operations are imperative, a use of them makes type systems hard to understand and prove [5]. In Section 4, we propose a declarative formulation of error-tolerant type systems.

3. Principal typing for debugging type errors

This section presents the basic idea of principal typing and shows how to put together variational typing and principal typing for debugging type errors.

3.1. Principal typing

In the Hindley-Milner type system (HM) and its implementation, the algorithm $\mathcal{W}$, type environment is an input. The type environment stores type information for free variables, and is updated accordingly as the inference algorithm traverses the program AST. As a result, type inference of the later part of the AST always depends on that of the earlier part.

Fig. 3 shows how $\mathcal{W}$ passes around and updates the type environment for typing the expression `rank` from Section 1. When first visiting the abstraction node, $\mathcal{W}$ assigns a fresh type variable α as the type to the parameter x. This type is updated to $\texttt{Char} \to \beta$ after visiting the node x `'1'`. The reference to x in x `True` will thus have the type $\texttt{Char} \to \beta$. It is easy to see that type inference is left-biased.

This bias hinders incremental type inference since even a small change in the left subtree will require almost the full type inference to be redone. For example, if we change `'1'` to `1` in Fig. 3 we have to update type information at all nodes.

In contrast, principal typing[2] [28] doesn't suffer from this bias. Type inference in principal typing is done bottom-up. At each variable reference, the variable is assumed to have a fresh type. The assumptions are refined as the inference gets closer to the root and are made consistent at the corresponding abstraction. As an example, let's again refer to Fig. 3, where dashed blue lines denote how assumptions flow for principal typing. We observe that the assumptions for the same variable but different occurrences flow upwards from leaves into the abstraction. As a result, there is no left-to-right bias.

Principal typing is more amenable to incremental typing. Again, assume `'1'` is changed to `1` in Fig. 3. We only need to update four nodes, the path from `'1'` to the root. We will design *e*CFT based on principal typing, which has already been used in incremental type checking [28,2,19] and smartest recompilation [45].

3.2. Variational typing and principal typing for debugging type errors

When an expression is ill typed, we generally ask two questions: Which subexpression caused the type error and how should we change the subexpression to remove the type error? Conceptually, *e*CFT (and CFT) addresses these problems in following steps: (1) assuming that all subexpressions may be the error causes, (2) computing the types that subexpression ought to have to remove the type error, and (3) finding real error causes by filtering out subexpressions the types they have

[2] Principal typing is very different from principal types.

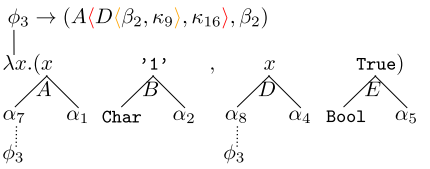

Fig. 4. Type error debugging for $\lambda x.(x\ 1, x\ \texttt{'a'})$ with variational typing and principal typing.

differing from those they ought to have. However, implementing this idea seems to have a high complexity: for each of all subexpressions and their possible combinations, we have to perform type inference of the expression to find out the type the subexpression ought to have. To combat with this high complexity, *e*CFT and CFT employ variational typing to reuse computations. Specifically, they create at each AST leaf a variational type, where the first alternative is the type of the leaf under normal type inference and the second alternative is the type the leaf ought to have to remove the type error in the whole expression. Our previous work [7] explained why CFT considered changing leaves only and how it achieved high precision.

We illustrate this idea by debugging type errors for `rank`. We use principal typing to facilitate incremental updates of error fixes as programs are changed. Fig. 4 shows choice types that are created for leaves. It also presents the result type of the whole expression.

Remember that type information flows bottom-up in principal typing, and thus we first type leaves. According to Fig. 4, the type created for the first x is $A\langle\alpha_7, \alpha_1\rangle$, where α_7 is the assumption we make for x in principal typing and α_1 represents the type that x ought to have in case x is identified as an error cause. The type for `'1'` under normal type inference is `Char`, thus the type created for `'1'` is $B\langle\texttt{Char}, \alpha_2\rangle$. Again, α_2 indicates how we need to change `'1'` to remove the type error.

As the inference algorithm moves up in the tree, fresh types are made more and more concrete as unification problems are solved.[3] For example, after visiting the application x `'1'`, α_7 will become $A\langle B\langle\texttt{Char}, \alpha_2\rangle \to \beta_1, \kappa_1\rangle$ and after visiting the abstraction, it will further be made concrete to ϕ_3, which is shown below. After the type inference is completed, the concrete types that variables are mapped to are as follows. The inference algorithm also returns the typing pattern $\pi = A\langle B\langle D\langle E\langle \bot, \top\rangle, \top\rangle, \top\rangle, \top\rangle$. The detailed typing process is given in Fig. 8.

$$
\begin{aligned}
\alpha_1 &= A\langle \kappa_2, B\langle \texttt{Char}, \kappa_{15}\rangle \to \kappa_{16}\rangle \\
\alpha_2 &= A\langle B\langle \kappa_{13}, D\langle E\langle \texttt{Bool}, \kappa_{12}\rangle, \kappa_{11}\rangle\rangle, \kappa_{15}\rangle \\
\alpha_4 &= D\langle \kappa_4, E\langle \texttt{Bool}, A\langle B\langle \kappa_7, \kappa_{12}\rangle, \kappa_{14}\rangle\rangle \to \beta_2\rangle \\
\alpha_5 &= A\langle B\langle D\langle \texttt{Char}, \kappa_7\rangle, \kappa_{12}\rangle, \kappa_{14}\rangle \\
\phi_3 = \alpha_7 = \alpha_8 &= A\langle B\langle \texttt{Char}, D\langle E\langle \texttt{Bool}, \kappa_{12}\rangle, \kappa_{11}\rangle\rangle \to D\langle \beta_2, \kappa_9\rangle, \\
&\qquad D\langle E\langle \texttt{Bool}, \kappa_{14}\rangle \to \beta_2, \kappa_3\rangle\rangle \\
\beta_1 &= A\langle D\langle \beta_2, \kappa_9\rangle, \kappa_{16}\rangle
\end{aligned}
$$

Given the 4 choices that are generated during the type inference, we can have 16 potential decisions with each decision takes either the first or second alternative of every choice. For example, the decision $\delta_1 = \{A.1, B.1, D.1, E.1\}$ includes the first alternatives of all choices. The decision specifies how we remove the type error. If the decision includes $d.1$, it means that we don't change the subexpression that d is created for. Otherwise, if it includes $d.2$, it means that we change the corresponding subexpression.

We consider two decisions for removing the type error, δ_1 from the previous paragraph and $\delta_2 = \{A.1, B.2, D.1, E.1\}$. For δ_1, $\lfloor \pi \rfloor_{\delta_1} = \bot$, which means that the result is invalid if we don't make any change to the original expression. In other words, the original expression is ill typed. For δ_2, we have $\lfloor \pi \rfloor_{\delta_2} = \top$, which means that the result is valid. In other words, if we change `'1'` (what B corresponds to), then the resulting expression is well typed. Furthermore, if we change it to something of type $\lfloor \alpha_2 \rfloor_{\delta_2} = \texttt{Bool}$, the resulting expression has the type $\lfloor \phi_3 \to (A\langle D\langle \beta_2, \kappa_9\rangle, \kappa_{16}\rangle, \beta_2)\rfloor_{\delta_2} = (\texttt{Bool} \to \beta_2) \to (\beta_2, \beta_2)$. Overall, we can extract the error message: "If we change `'1'` to something of type `Bool`, then the corrected expression has the type $(\texttt{Bool} \to \beta_2) \to (\beta_2, \beta_2)$".

For this expression, there are 15 possible ways to fix the type error, including 4 that change only one of the four leaves, respectively. CFT [7] includes a set of heuristics to rank these error fixes to present the most likely ones to the user first. For this expression, the first two suggested fixes are changing `True` to some expression that has the type `Char` or change `'1'` to something that has the type `Bool`. The next two suggestions change either occurrence of `x`. We will use the same set of heuristics in *e*CFT.

[3] For example, solving the unification problem between a type variable and the type `Int` makes the type variable to be `Int`, which is more concrete than the type variable. We talk more about unification in Section 6.

4. Declarative error-tolerant variational typing

Early work on error-tolerant variational typing (Section 2.3) directly introduces error types to the type syntax and carefully generates and propagates type errors during the typing process [12]. Instead of introducing error types into the type syntax to represent type errors explicitly, we propose to use typing patterns to indicate which variants of the typing result are correct and which are incorrect. More formally, the new type judgment has the form $\pi; \Gamma \vdash e : \phi$, meaning that e has the type ϕ under Γ with the validity restriction π. Intuitively, π specifies that only the variants that π contains $\top$s are valid (type correct) and those that π contains $\bot$s are invalid (type incorrect) and shouldn't be trusted.

For example, we have $\top; \Gamma \vdash \texttt{1} : \texttt{Int}$, saying that `1` always has the type `Int`. Similarly, we have $\bot; \Gamma \vdash \texttt{True} : \texttt{Int}$, which says that the typing result `True` has the type `Int` can not be trusted, as indicated by the $\bot$ at the beginning of the judgment. However, $\top; \Gamma \vdash \texttt{True} : \texttt{Int}$ is not a valid judgment because it says that the expression `True` has the type `Int` can be trusted. Similarly, $B\langle \top, \bot \rangle; \Gamma \vdash B\langle \texttt{1}, \texttt{True} \rangle : \texttt{Int}$ is a valid judgment because it encodes two judgments (obtained by selecting the pattern $B\langle \top, \bot \rangle$, the expression $B\langle \texttt{1}, \texttt{True} \rangle$, and the type `Int` by the selectors $B.1$ and $B.2$) $\top; \Gamma \vdash \texttt{1} : \texttt{Int}$ and $\bot; \Gamma \vdash \texttt{True} : \texttt{Int}$, and both of them are valid. Note that the π is an input in the judgment. In type inference, however, π will be computed through unification. We discuss this in Section 6.

With the extended judgment form, the rule for typing function applications can be formalized as follows.

$$\frac{\pi; \Gamma \vdash e_1 : \phi_1 \qquad \pi; \Gamma \vdash e_2 : \phi_2 \qquad \phi_1 \equiv_\pi \phi_2 \to \phi}{\pi; \Gamma \vdash e_1\ e_2 : \phi}\ \textsc{T-App-Decl} \qquad \frac{\forall \delta : \lfloor \pi \rfloor_\delta = \top \Rightarrow \lfloor \phi_1 \rfloor_\delta \equiv \lfloor \phi_2 \rfloor_\delta}{\phi_1 \equiv_\pi \phi_2}$$

The typing rule can be read as: if the function e_1 has the type ϕ_1, the argument e_2 has the type ϕ_2, and ϕ_1 is equivalent to $\phi_2 \to \phi$, all under the validity restriction π, then the application $e_1\ e_2$ has the type ϕ under the same π.

The typing pattern π in $\phi_1 \equiv_\pi \phi_2$ specifies where ϕ_1 and ϕ_2 need to be equivalent. Specifically, for any δ, if $\lfloor \pi \rfloor_\delta = \top$, then $\lfloor \phi_1 \rfloor_\delta \equiv \lfloor \phi_2 \rfloor_\delta$. However, if $\lfloor \pi \rfloor_\delta = \bot$, then $\lfloor \phi_1 \rfloor_\delta$ and $\lfloor \phi_2 \rfloor_\delta$ do not have to be equivalent. For example, all $\texttt{Int} \equiv_\top \texttt{Int}$, $\texttt{Int} \equiv_\top A\langle \texttt{Int}, \texttt{Int} \rangle$, $\texttt{Int} \equiv_\bot \texttt{Bool}$ (note the subscript $\bot$) are all valid, but $\texttt{Int} \equiv_\top \texttt{Bool}$ is invalid.

Both rules T-App-Err (Section 2.3) and T-App-Decl handles error-tolerant variational typing. The rule T-App-Err involves three operations: type lifting $\uparrow$, type matching $\bowtie$, and type masking $\lhd$. These operations make the rule T-App-Err operational. On the other hand, the rule T-App-Decl gets rid of these three operations. We observe that the rule T-App-Decl bears more resemblance to the standard typing rule for application T-App-Std (Section 2.2) than the rule T-App-Err does. More importantly, the idea and the machinery we developed in the rule T-App-Decl simplifies the presentation of our later typing rules in Sections 5.2 and 5.3. In particular, had we used the idea from T-App-Err to handle error tolerance there, we need to replace each occurrence of π in these rules with the three operations from T-App-Err, which would significantly clutter our typing rules later.

Based on the rule T-App-Decl, we can type $A\langle \texttt{succ}, \texttt{odd} \rangle\ B\langle \texttt{1}, \texttt{True} \rangle$ as follows.

$$\begin{aligned} \text{Premises} &\begin{cases} B\langle \top, \bot \rangle; \varnothing \vdash A\langle \texttt{succ}, \texttt{odd} \rangle : \texttt{Int} \to A\langle \texttt{Int}, \texttt{Bool} \rangle \\ B\langle \top, \bot \rangle; \varnothing \vdash B\langle \texttt{1}, \texttt{True} \rangle : \texttt{Int} \\ \texttt{Int} \to A\langle \texttt{Int}, \texttt{Bool} \rangle \equiv_{B\langle \top, \bot \rangle} \texttt{Int} \to A\langle \texttt{Int}, \texttt{Bool} \rangle \end{cases} \\ \text{Conclusion} &\begin{cases} B\langle \top, \bot \rangle; \varnothing \vdash A\langle \texttt{succ}, \texttt{odd} \rangle\ B\langle \texttt{1}, \texttt{True} \rangle : A\langle \texttt{Int}, \texttt{Bool} \rangle \end{cases} \end{aligned}$$

Note that the pattern $B\langle \top, \bot \rangle$ is an input to the typing process, which means that we need to come up with this before we can begin the process. Fortunately, in the type system implementation, the pattern for each expression becomes an output of the type inference result. The idea of providing some information as input in declarative specification while compute it in type system implementation is standard in type system research. One such example is giving types to parameters for abstractions [40].

The result indicates that the expression $A\langle \texttt{succ}, \texttt{odd} \rangle\ B\langle \texttt{1}, \texttt{True} \rangle$ has the type $A\langle \texttt{Int}, \texttt{Bool} \rangle$. However, notice the pattern $B\langle \top, \bot \rangle$, which indicates that the result is valid in $B.1$ (because of the $\top$ in $B\langle \top, \bot \rangle$ in $B.1$) but not in $B.2$ (because of the $\bot$). In Section 2.3, we have seen that the expression $A\langle \texttt{succ}, \texttt{odd} \rangle\ B\langle \texttt{1}, \texttt{True} \rangle$ has the type $B\langle A\langle \texttt{Int}, \texttt{Bool} \rangle, \bot \rangle$ under the rule T-App-Err. We can observe that the typing results by T-App-Err and T-App-Decl are the same. In general, if we have $\pi; \Gamma \vdash e : \phi_1$ with T-App-Decl and $\Gamma \vdash e : \phi_2$ with T-App-Err, then $\phi_2 = \pi \lhd \phi_1$.

5. Type system

In this section, we present the type system based on principal typing for producing a complete set of error fixes. The syntax is given in Section 5.1, basic typing rules are discussed in Section 5.2, and the top-level rule for handling unbound variables is given in Section 5.3. A variable is unbound if it is used without being declared. For example, in the expression $\lambda x.x\ y$, the x in the body is bound but the y is unbound.

Term variables	x, y, z	Value constants	c
Type variables	α, β, κ	Type constants	γ
Choice names	A, B, D	Choice meta variable	d
Alternative index	$i \in \{1, 2\}$	Program locations	l

Expressions $e, f ::= c \mid x \mid \lambda x.e \mid e\ e \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid$
$\textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e$

Monotypes	$\tau ::= \gamma \mid \alpha \mid \tau \to \tau$
Variational types	$\phi ::= \tau \mid d\langle\phi, \phi\rangle \mid \phi \to \phi$
Typing patterns	$\pi ::= \bot \mid \top \mid d\langle\pi, \pi\rangle$
Selectors	$s ::= d.i$

Type assumption sets	$\mathcal{A} ::= \varnothing \mid \mathcal{A}, (x, \phi, d)$
Substitutions	$\theta ::= \varnothing \mid \theta, \alpha \mapsto \phi$
Choice environments	$\Delta ::= \varnothing \mid \Delta, (l, d\langle\phi, \phi\rangle)$

Fig. 5. Syntax of expressions, types, and environments.

5.1. Syntax

The syntax for types, expressions, and meta environments is given in Fig. 5. We support conditionals through **if** expressions and support polymorphism through **let** expressions.

We stratify the type definition into two layers. First, monotypes, ranged over by τ, include constant types (γ), type variables (α), and function types. In addition to α, we use β to denote fresh type variables used for representing function application results and use κ to denote fresh type variables generated during unification. Variational types extend monotypes with choice types. We support polymorphism without explicitly defining polymorphic types by using a method proposed in [27, §4].

We use $\overline{o}$ to denote a sequence of objects $o_1, \ldots, o_n$ for any object o. The function *TVs* computes the set of type variables in types. It collects type variables from both alternatives for choice types and has conventional definition for other types. We use $\theta(\phi)$ to denote the application of a type substitution. It replaces free type variables in ϕ by the corresponding images in θ. Its definition is standard except for choice types, where substitution applies to both alternatives.

An assumption set $\mathcal{A}$ maps expression variables to variational types. Moreover, $\mathcal{A}$ stores the corresponding choice name of each variable reference. We write $\mathcal{A}^T(x)$ and $\mathcal{A}^D(x)$ to get the set of types and choice names that x maps to in $\mathcal{A}$, respectively.

We use l to represent program locations and use the function $\ell_e(f)$ to return the location of f in e. We may omit the subscript e when the context is clear. We assume f uniquely determines the location. The exact definition of $\ell(\cdot)$ doesn't matter.

The choice environment Δ associates each leaf l with a choice type $d\langle\phi_1, \alpha_2\rangle$ during the typing process. Note that ϕ_1 is the type under normal type inference for the subexpression at l and ϕ_2 is the alternative type for the same subexpression to remove the type error.

5.2. Basic typing rules

We present the typing rules in Fig. 6. The typing judgment has the form $\pi; \mathcal{A} \vdash e : \phi | \Delta$, meaning that the expression e has the type ϕ with the assumption set $\mathcal{A}$, the validity restriction π, and the change information Δ.

The conditions "d is fresh" in rules Con and Var are always satisfied since we have an unlimited supply of choice names. The rule Con says that if c is of the type γ, then in our type system it has the type $d\langle\gamma, \phi\rangle$ to indicate that we can change c to any type ϕ to remove the type error in case c is an error cause. No assumption is made for typing c, and thus the $\mathcal{A}$ component is empty. The choice environment records the change as $\{(\ell(c), d\langle\gamma, \phi\rangle)\}$. The function $\ell(c)$, defined in Section 5.1, returns the location of c in the program. This function call is needed because the Δ is a mapping from locations to types. For this rule, we can use any π since the typing of the constant is always valid.

The rule Var for variables is similar to the rule Con. The only difference is that the variable may have any type ϕ and the assumption is recorded in $\mathcal{A}$. At variable references, we always assume variables are bound. Therefore the typing pattern component can be any value. We deal with unbound variables in Section 5.3.

Given an abstraction $\lambda x.e$, we first type the body e, which may contain multiple assumptions for the parameter x. These assumptions need to be consistent for the abstraction to be well typed. The second premise in the Abs rule ensures that all assumptions are equivalent to each other with the restriction π. The meaning of the notation $\equiv_\pi$ is given earlier in Section 4 (below the T-App-Decl rule). Several examples are also given there. Intuitively, in $\phi_1 \equiv \phi_2$, ϕ_1 and ϕ_2 are equivalent or equal (syntactically the same) when selected with any decision. However, in $\phi_1 \equiv_\pi \phi_2$, ϕ_1 and ϕ_2 are equivalent or equal (syntactically the same) only when selected with a decision δ such that $\lfloor\pi\rfloor_\delta = \top$. The second premise thus says that ϕ_1 is the parameter type if it is equivalent to all the assumptions made about the parameter when typing the body, under the restriction π. The assumptions for the abstraction are the assumptions for its body minus those for the parameter x.

$$\boxed{\pi ; \mathcal{A} \vdash e : \phi | \Delta}$$

$$\textsc{Con}\ \frac{c \text{ is of type } \gamma \qquad d \text{ is fresh}}{\pi ; \varnothing \vdash c : d\langle \gamma, \phi \rangle | \{(\ell(c), d\langle \gamma, \phi \rangle)\}}$$

$$\textsc{Var}\ \frac{d \text{ is fresh}}{\pi ; \{(x, \phi)\} \vdash x : d\langle \phi, \phi_1 \rangle | \{(\ell(x), d\langle \phi, \phi_1 \rangle)\}}$$

$$\textsc{Abs}\ \frac{\pi ; \mathcal{A} \vdash e : \phi | \Delta \qquad \phi_1 \equiv_\pi (\phi_i \mid \forall \phi_i \in \mathcal{A}^T(x))}{\pi ; \mathcal{A} \setminus x \vdash \lambda x.e : \phi_1 \to \phi | \Delta}$$

$$\textsc{App}\ \frac{\pi ; \mathcal{A}_1 \vdash e_1 : \phi_1 | \Delta_1 \qquad \pi ; \mathcal{A}_2 \vdash e_2 : \phi_2 | \Delta_2 \qquad \phi_1 \equiv_\pi \phi_2 \to \phi}{\pi ; \mathcal{A}_1 \cup \mathcal{A}_2 \vdash e_1\ e_2 : \phi | \Delta_1 \cup \Delta_2}$$

$$\textsc{If}\ \frac{(\pi ; \mathcal{A}_i \vdash e_i : \phi_i | \Delta_i)^{i:1..3} \qquad \texttt{Bool} \equiv_\pi \phi_1 \qquad \phi_2 \equiv_\pi \phi_3}{\pi ; \mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3 \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \phi_2 | \Delta_1 \cup \Delta_2 \cup \Delta_3}$$

$$\textsc{Let}\ \frac{\pi ; \mathcal{A}_1 \vdash e_1 : \phi' | \Delta_1 \qquad \pi ; \mathcal{A}_2 \vdash e_2 : \phi'' | \Delta_2 \qquad \overline{\alpha} = TVs(\phi') - TVs(\mathcal{A}_1) \qquad \phi' \equiv_\pi^{\overline{\alpha}} (\phi_i | \forall \phi_i \in \mathcal{A}_2^T(x))}{\pi ; \mathcal{A}_1 \cup (\mathcal{A}_2 \setminus x) \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \phi'' | \Delta_1 \cup \Delta_2}$$

Fig. 6. Basic typing rules.

The choice environment of the abstraction is the same as that for its body since we don't consider changes in function parameters.

The App rule is very similar to the application rule discussed in Section 4. The only difference is that here we have to merge assumption sets from the subexpressions and also change environments from them. The If rule is straightforward: it ensures the type of the condition is equivalent to `Bool` and the branches have the equivalent type, both with the restriction π. The conditional inherits assumption sets and change environments from all its subexpressions.

In HM, we type **let** $x = e_1$ **in** e_2 by first tying e_1, binding x to the generalized type of e_1 in the type environment, and typing the expression e_2, where each access to x instantiates the type it is bound to. We observe that the body can't be typed without knowing the type of the variable x. How should we type **let** expressions in principal typing without introducing this dependency? We address this by requiring that all assumptions about x in e_2 be type instances of the type of e_1.

We formalize this idea in rule Let. The rule uses an auxiliary relation of the form $\phi_1 \equiv_\pi^{\overline{\alpha}} \phi_2$, which states that ϕ_2 is an instance of ϕ_1 with the restriction π by instantiating variables in $\overline{\alpha}$. The relation is satisfied if the following condition holds. The notation $\equiv_\pi$ is defined in Section 4 (below the rule T-App-Decl).

$$\exists \theta : \theta(\phi_1) \equiv_\pi \phi_2 \text{ where } dom(\theta) = \overline{\alpha}$$

Other than this relation, the rule is quite standard.

5.3. Handling unbound variables

In rule Var, we assume that all variables are bound. However, this is not always the case. A variable is bound if it was declared and is unbound otherwise. For example, in the expression $\lambda x.x\ y$, x is bound and y is unbound. In standard typing [40], we first visit the abstraction then visit its body, allowing us to detect unbound variables once we encounter them. However, in principal typing (the typing mechanism used in this paper), we first visit the body and then the abstraction itself, and thus we are not able to determine whether variables are bound when we encounter them. We need to wait until we have typed the whole expression.

How do we determine if an expression contains unbound variables after typing it? by checking $\mathcal{A}$. If $\mathcal{A}$ is empty, then all variables are bound. Otherwise, variables that have binding information left in $\mathcal{A}$ are unbound.

What should we do with unbound variables? Or more specifically, what does it mean if (x, ϕ, d) still belongs to $\mathcal{A}$ after typing the expression e with $\pi ; \mathcal{A} \vdash e : \phi | \Delta$? This means that the access to x should be incorrect but when typing e we assumed it was correct under the Var rule. We can address this problem by adjusting the validity restriction π. In fact, (x, ϕ, d) still belongs to $\mathcal{A}$ means that the typing result is invalid in $d.1$. Thus, we need to worsen π by $d\langle \bot, \top \rangle$ to keep the typing result valid.

In fact, there is a simpler way to address this problem. The key observation here is that if the typing pattern π is already worse than $d\langle \bot, \top \rangle$, then we don't need to worsen it anymore. Before presenting the typing rule based on this idea, we first formalize the notion that a typing pattern (π_1) is *worse* than the other one (π_2), written as $\pi_1 \le \pi_2$, as follows.

$$\pi \le \top \qquad \bot \le \pi \qquad \frac{\pi_1 \le \pi_2 \qquad \pi_2 \le \pi_3}{\pi_1 \le \pi_3} \qquad \frac{\pi_1 \equiv \pi_2}{\pi_1 \le \pi_2} \qquad \frac{\pi_1 \le \pi_3 \qquad \pi_2 \le \pi_4}{d\langle \pi_1, \pi_2 \rangle \le d\langle \pi_3, \pi_4 \rangle}$$

$$
\begin{array}{l}
\mathit{infer1} : e \to \pi \times \mathcal{A} \times \phi \times \Delta \\
\text{(1a) } \mathit{infer1}(x) = \\
\quad \phi \leftarrow d\langle \alpha_1, \alpha_2 \rangle \qquad \{\text{- } d, \alpha_1, \text{ and } \alpha_2 \text{ fresh-}\} \\
\quad \text{return } (\top, \{(x, \alpha_1, d)\}, \phi, \{(\ell(x), \phi)\}) \\
\text{(1b) } \mathit{infer1}(e_1\ e_2) = \\
\quad (\pi_1, \mathcal{A}_1, \phi_1, \Delta_1) \leftarrow \mathit{infer1}(e_1) \\
\quad (\pi_2, \mathcal{A}_2, \phi_2, \Delta_2) \leftarrow \mathit{infer1}(e_2) \\
\quad (\pi, \theta) \leftarrow \mathit{vunify}(\phi_1, \phi_2 \to \beta) \qquad \{\text{-}\beta \text{ fresh-}\} \\
\quad \text{return } (\pi \otimes \pi_1 \otimes \pi_2, \theta(\mathcal{A}_1 \cup \mathcal{A}_2), \theta(\beta), \theta(\Delta_1 \cup \Delta_2)) \\
\text{(1c) } \mathit{infer1}(\lambda x.e) = \\
\quad (\pi, \mathcal{A}, \phi, \Delta) \leftarrow \mathit{infer1}(e) \\
\quad \phi_p \leftarrow \alpha;\ \theta \leftarrow \{\} \qquad \{\text{- } \alpha \text{ fresh-}\} \\
\quad \text{for each } \phi_i \text{ in } \mathcal{A}^T(x) \\
\quad\quad (\pi_i, \theta_i) \leftarrow \mathit{vunify}(\theta(\phi_p), \theta(\phi_i)) \\
\quad\quad \pi \leftarrow \pi \otimes \pi_i;\ \theta \leftarrow \theta_i \circ \theta \\
\quad \text{return } (\pi, \theta(\mathcal{A} \setminus x), \theta(\phi_p \to \phi), \theta(\Delta)) \\
\text{(1d) } \mathit{infer1}(\textbf{let } x = e_1 \textbf{ in } e_2) = \\
\quad (\pi_1, \mathcal{A}_1, \phi_1, \Delta_1) \leftarrow \mathit{infer1}(e_1) \\
\quad (\pi_2, \mathcal{A}_2, \phi_2, \Delta_2) \leftarrow \mathit{infer1}(e_2) \\
\quad \pi \leftarrow \pi_1 \otimes \pi_2 \\
\quad \overline{\alpha} \leftarrow \mathit{TVs}(\phi_1) - \mathit{TVs}(\mathcal{A}_1);\ \theta \leftarrow \{\} \\
\quad \text{for each } \phi_i \text{ in } \mathcal{A}_2^T(x) \\
\quad\quad (\pi_i, \theta_i) \leftarrow \mathit{vunify}(\{\overline{\alpha \mapsto \beta}\}(\theta(\phi_1)), \theta(\phi_i)) \qquad \{\text{-}\overline{\beta} \text{ fresh-}\} \\
\quad\quad \pi \leftarrow \pi \otimes \pi_i;\ \theta \leftarrow \theta_i \circ \theta \\
\quad \text{return } (\pi, \theta(\mathcal{A}_1 \cup (\mathcal{A}_2 \setminus x)), \theta(\phi_2), \theta(\Delta_1 \cup \Delta_2))
\end{array}
$$

Fig. 7. An inference algorithm that recomputes all error fixes.

Intuitively, $\pi_1 \leq \pi_2$ expresses that π_1 contains $\bot$s in more variants than π_2 does. The first two rules say that all typing patterns are worse than $\top$ and better than $\bot$. The third rule states that the relation is transitive. In the fourth rule, we reuse the machinery of type equivalence by interpreting $\bot$ and $\top$ as two constant types. The rule then says that two equivalent patterns satisfy the $\leq$ relation. The last rule states that two choice patterns satisfy $\leq$ if both their corresponding alternatives satisfy $\leq$.

With $\leq$, we can formalize the rule for unbound variables declaratively as follows.

$$\textsc{Unbound}\ \frac{\pi; \mathcal{A} \vdash e : \phi | \Delta \qquad \forall (x, \phi, d) \in \mathcal{A}\colon \pi \leq d\langle \bot, \top \rangle}{\pi \vdash_M e : \phi | \Delta}$$

We can see that $\mathcal{A}$ in the premise disappears in the conclusion. Intuitively, the rule says that if π is worse enough, then the residual assumptions can simply be forgotten. The subscript M indicates that this is the main rule on top of all other typing rules. In fact, given an expression, we should use this rule to compute error fixes.

As an example, consider the expression $ubx = x\ \texttt{1}$. We have $\top; \{(x, \phi, A)\} \vdash ubx : \texttt{Bool} | \Delta$, where $\phi = B\langle \texttt{Int}, \alpha_3 \rangle \to \texttt{Bool}$ and $\Delta = \{(\ell(x), \phi), (\ell(\texttt{1}), B\langle \texttt{Int}, \alpha_3 \rangle)\}$. Since $\top \not\leq A\langle \bot, \top \rangle$, we can't derive $\top \vdash_M ubx : \texttt{Bool} | \Delta$. Similarly, we have $A\langle \bot, \top \rangle; \{(x, \phi, A)\} \vdash ubx : \texttt{Bool} | \Delta$. Since $A\langle \bot, \top \rangle \leq A\langle \bot, \top \rangle$, we have $A\langle \bot, \top \rangle \vdash_M ubx : \texttt{Bool} | \Delta$, which tells us that without changing x, the expression is ill typed.

Although our type system is based on principal typing, it generates the same set of error fixes as CFT does, as captured in the following theorem.

Theorem 1 *(CFT equivalence).* $\pi; \varnothing \vdash e : \phi | \Delta \Leftrightarrow \varnothing \vdash_{CFT} e : \phi^{\bot} | \Delta^{\bot}$ *such that* $\phi \equiv_\pi \phi^{\bot}$, $\Delta \equiv_\pi \Delta^{\bot}$, *and* $\forall \delta : \lfloor \phi^{\bot} \rfloor_\delta \neq \bot \Leftrightarrow \lfloor \pi \rfloor_\delta \neq \bot$.

In the theorem, we use $\vdash_{CFT}$ to denote the typing relation of CFT. The superscript $\bot$ in $\phi^{\bot}$ and $\Delta^{\bot}$ reflects that errors types are embedded in the type syntax in CFT. The relation $\Delta \equiv_\pi \Delta^{\bot}$ is defined as $\forall l \in \mathit{dom}(\Delta)\colon \Delta(l) \equiv_\pi \Delta^{\bot}(l)$.

This theorem could be proved through an additional typing relation called type-update system [7, §4.3]. We can prove that both CFT and this type system are equivalent to the type-update system through inductions over typing relations. The proof of this theorem is given in the companion [11] of this paper.

6. Recompute all error fixes

We will present three type inference algorithms from this section through Section 8. This section develops an inference algorithm that recomputes all type error fixes as programs are updated, Section 7 shows a coarse incremental inference algorithm that reuses results so that only nodes that are affected by the change are retyped, and Section 8 develops a refined incremental inference algorithm on top of incremental variational unification.

As the type system in Section 5 is divided into two layers, the type inference algorithm follows a similar structure. The algorithm *infer1* in Fig. 7 provides a unification-based implementation of the typing rules from Fig. 6.

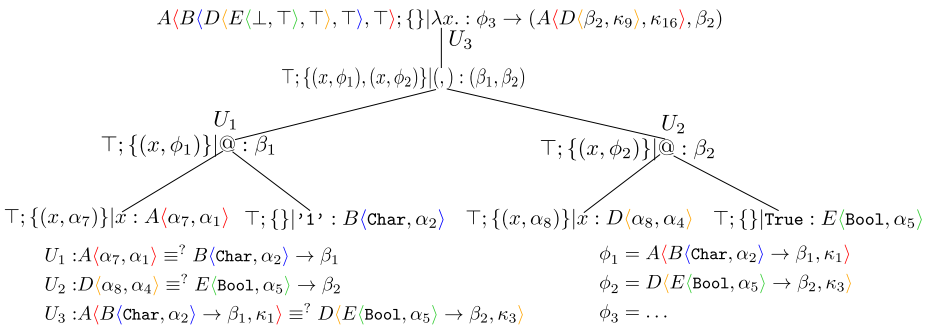

Fig. 8. Compute error fixes for `rank` with *infer1*. The value of ϕ_3 and other typing information can be found in Section 3.2.

The function *infer1* has the type $e\to\pi\times\mathcal{A}\times\phi\times\Delta$, that is, it takes in an expression and returns a typing pattern indicating which alternatives of the inference result are valid, an assumption set $\mathcal{A}$, a result type, and a choice environment storing type change information of leaves. Note that we don't need to return a unifier in *infer1* since we can always keep the typing assumption $\mathcal{A}$ that is returned by *infer1* up to date. Since the main logic for constants is very similar to that for variables and that for conditionals is very similar to that for applications, we have left out the cases for constants and conditionals in *infer1*.

We now briefly go through *infer1*. The case (1a) deals with variable references. As mentioned earlier, we always assume variables are bound at variable references. Therefore, *infer1* returns $\top$ for π, indicating no errors have occurred. Note that we need two fresh type variables α_1 and α_2 in (1a), where α_1 denotes that the variable can be mapped to any type based on its uses and α_2 denotes that the variable can be changed to something of type α_2 to remove the type error if the variable is an error cause. The main difference between them is that α_1 is recorded in $\mathcal{A}$ and will be later unified with other assumptions for the same variable.

An important difference between this case and rule Var (Fig. 6) for handling π is that in this case the returned π is $\top$ whereas in Var the π can be any pattern (if a symbol appears unconstrained in a typing rule, then it can be instantiated with any instance from the corresponding syntactical category). The reason we used π in Var is to simplify the type system specification: essentially we express that any π will make the conclusion in Var valid. In implementation, however, we aim to compute the "best" π in the sense that we should never introduce more errors than needed. Since typing a variable reference can always be succeeded, we return $\top$, the best pattern among all possible patterns. We will return to this discussion after the case (1b) below.

We use Fig. 8 for typing `rank` as a running example to illustrate *infer1*. In the tree, each node has a label in the form of $\pi;\mathcal{A}|e:\phi$, meaning that *infer1* infers π, $\mathcal{A}$, and ϕ for e. For simplicity, we omitted Δ in the figure. The case (1a) is used to compute the types for both occurrences of x in the figure.

Case (1b) deals with function applications. It first computes the results for the function and the argument independently and then unifies the function type and the argument type with *vunify*, a variational unification algorithm from [9]. *vunify* delegates unification problems involving plain types (types without variations) to the classical Robinson's unification algorithm [41], with, roughly, extensions in two dimensions.

First, *vunify* deals with variations. When variations are encountered in *vunify*, unification goes to their alternatives. For example, for the problem *vunify*$(A\langle\texttt{Int},\alpha_1\rangle,\texttt{Int})$, the variation $A\langle\texttt{Int},\alpha_1\rangle$ is broken down into `Int` and α_1, yielding two subproblems: *vunify*$(\texttt{Int},\texttt{Int})$ and *vunify*$(\alpha_1,\texttt{Int})$. The first subproblem is simply discharged because the two types being unified are the same. For the second subproblem, the solution is mapping α_1 to `Int`. However, as α_1 appeared in the second alternative of the choice A only (in the original problem), there is no constraint regarding α_1 in the first alternative of A. As a result, the final solution to α_1 is $A\langle\kappa,\texttt{Int}\rangle$, where κ is a fresh type variable, indicating that the result for α_1 is unconstrained in the first alternative of A.

Second, while Robinson's algorithm terminates the unification process if it encounters a constraint that can not be solved, *vunify* will keep on constraint solving and use a $\bot$ to denote where constraint solving failed. For example, consider solving *vunify*$(A\langle\texttt{Bool},\alpha_1\rangle,\texttt{Int})$, a slightly adapted version of the example from the last paragraph. Again, for this problem, $A\langle\texttt{Bool},\alpha_1\rangle$ is broken down into `Bool` and α_1. Thus, the first subproblem will be *vunify*$(\texttt{Bool},\texttt{Int})$, which could not be solved because `Bool` and `Int` are different. However, in *vunify*, we do not terminate the solving process, but use a $\bot$ to indicate that the unification problem could not be solved. The second subproblem is the same as in the last paragraph and could be solved successfully with the same solution. In addition to returning the unifier, *vunify* returns a $\top$ to indicate constraint solving is successful. In general, *vunify* returns a pair (π,θ), where π is the returned pattern and θ is a unifier. For *vunify*$(A\langle\texttt{Bool},\alpha_1\rangle,\texttt{Int})$, the result is $(A\langle\bot,\top\rangle,\{\alpha_1\mapsto A\langle\kappa,\texttt{Int}\rangle\})$. In Section 4, we mentioned that the typing pattern π is an input in declarative specification and is an output in type inference. In fact, the π is returned from *vunify*.

For *vunify*$(A\langle\texttt{Int},\alpha_1\rangle,B\langle\texttt{Bool},\alpha_2\rangle)$, the result is:

$$(A\langle B\langle\bot,\top\rangle,\top\rangle,\{\alpha_1\mapsto A\langle\kappa_3,B\langle\texttt{Bool},\kappa_2\rangle\rangle,\alpha_2\mapsto A\langle B\langle\kappa_1,\texttt{Int}\rangle,\kappa_2\rangle\})$$

The typing pattern component tells us that only one alternative, identified by $\{A.1,B.1\}$, fails to unify.

In the rule App (Fig. 6), the same π appears four times, meaning that in all premises and the conclusion the typing pattern should be the same. This handling of patterns significantly simplifies this typing rule, as discussed in Section 4. In type inference, however, different subexpressions will return different "best" typing patterns, and we need to combine them together to return the "best" pattern for typing the application itself. Theorem 3 below states that the typing pattern returned from type inference is indeed the best with respect to $\leq$.

Specifically, we have three tying patterns, π_1 and π_2 from typing e_1 and e_2, respectively, and π from *vunify*. We will combine them together through the $\otimes$ operation, defined as follows (The same as in Section 2.3, reproduced here for readability purposes.).

$$\bot \otimes \pi = \bot \qquad \top \otimes \pi = \pi \qquad d\langle \pi_1, \pi_2 \rangle \otimes \pi = d\langle \pi_1 \otimes \pi, \pi_2 \otimes \pi \rangle$$

The result of $\otimes$ is $\top$ only if both operands are $\top$. Intuitively, $\otimes$ can be understood as the logical and operation if we interpret $\top$and $\bot$ as the truth values true and false, respectively. Based on this operation, *infer1* returns $\pi \otimes \pi_1 \otimes \pi_2$, meaning that e_1 e_2 is well typed only in the alternatives that both e_1 and e_2 are correct and the argument type of e_1 unifies with the type of e_2.

In Fig. 8, each @ represents an application and thus the corresponding node is typed with (1b). Typing each node needs to solve a unification problem that is attached to the node. The figure also includes a node that uses a pair constructor $(,)$. The rule for typing this constructor is standard and is omitted in this paper.

The case (1c) for typing abstractions is more complicated since we have to remove assumptions about the bound variable from $\mathcal{A}$. This is done through a for loop with the help of the variables ϕ_p for representing the parameter type of the function and θ for accumulating the unifiers returned in the loop. In each iteration, *vunify* is used to unify two assumptions for the parameter. This case returns a typing pattern that has a $\top$ in each alternative that e is well typed and all assumptions are consistent.

In Fig. 8, there are two assumptions ϕ_1 and ϕ_2 for the variable x before reaching the abstraction, the root node of the tree. They are made consistent by solving the unification problem $\phi_1 \equiv^? \phi_2$. The result of typing this expression is presented in Section 3.2, where we also showed how to extract error messages based on the result.

The case (1d) types polymorphic **let** expressions. Although we don't represent polymorphic types explicitly in the paper, we support polymorphism in principal typing through an approach proposed in [27]. The main logic of (1d) is very similar to that of (1c) and we will not discuss it in detail. The notation $\{\overline{\alpha \mapsto \beta}\}$ creates a substitution that maps type variables $\alpha_1, \alpha_2, \ldots, \alpha_n$ to fresh type variables $\beta_1, \beta_2, \ldots, \beta_n$, respectively. The uses of fresh type variables βs help implement polymorphic polymorphism [27]. The operation $\theta_i \circ \theta$ composes two unifiers θ_i and θ, a commonly used operation in type inference. Its definition is $\theta_i \circ \theta = \theta_i \cup \{\alpha \mapsto \theta_i(\phi) \mid \alpha \mapsto \phi \in \theta\}$.

In typing specification, we used the rule Unbound to handle unbound variables, we need an implementation of this rule in type inference. While π is an input in Unbound, here π is an output so we have to compute it. We realize this by worsening the typing pattern resulted from typing the expression with *infer1*. Specifically, for each entry (x, ϕ, d) remained in $\mathcal{A}$, the typing information that can be reached from $d.1$ contains errors. The following function *inferM* implements this idea. This function is also the main entry of the inference algorithm.

$inferM : e \to \pi \times \phi \times \Delta$

inferM(e)
  $(\pi, \mathcal{A}, \phi, \Delta) \leftarrow$ *infer1*(e)
  for each (x, ϕ, d) in $\mathcal{A}$
    $\pi \leftarrow \pi \otimes d\langle \bot, \top \rangle$
  return (π, ϕ, Δ)

Consider again the expression *ubx* from Section 5.3. *infer1*$(ubx) = (\top, \{(x, A\langle \phi, \kappa \rangle, A)\}, \beta_1, \Delta)$ and *inferM*$(ubx) = (A\langle \bot, \top \rangle, \beta_1, \Delta)$, where $\phi = B\langle \texttt{Int}, \alpha_3 \rangle \to \beta_1$ and $\Delta = \{(\ell(x), \phi), (\ell(1), B\langle \texttt{Int}, \alpha_3 \rangle)\}$.

We now investigate the properties of *infer1* and *inferM*. First, both of them are sound with respect to the typing relations discussed in Section 5.

Theorem 2 *(Inference soundness). Given e, if infer1*$(e) = (\pi, \mathcal{A}, \phi, \Delta)$*, then* $\mathcal{A}; \pi \vdash e : \phi | \Delta$*. Similarly, if inferM*$(e) = (\pi, \phi, \Delta)$*, then* $\pi \vdash_M e : \phi | \Delta$.

It's likely that in $\mathcal{A}$ we have multiple assumptions for some variable x, which means that different uses of x have different types. Here we require that the variable reference at the same location has the same type in *infer1* and the typing relation in Fig. 6. This theorem can be proved with an induction over different cases in Fig. 7. The proof of this and next theorem is given in the companion [11] of this paper.

The *infer1* and *inferM* are also complete, principal, and introducing fewest errors, as captured in the following theorem.

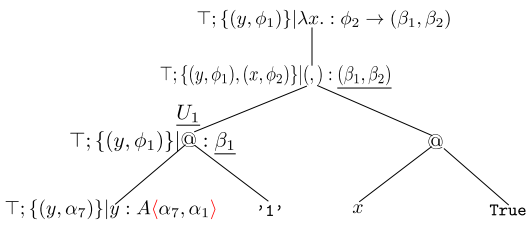

Fig. 9. Update typing information for changing the first occurrence of `x` in `rank` to `y`. The values for underlined objects are reused. The values of U_1, ϕ_1, and ϕ_2 are the same as those in Fig. 8 and are omitted in this figure.

Theorem 3 *(Inference completeness, principality, and error-minimality). If* $\pi_1; \mathcal{A}_1 \vdash e : \phi_1 | \Delta_1$, *then* $\mathit{infer1}(e) = (\pi_2, \mathcal{A}_2, \phi_2, \Delta_2)$ *(*completeness*) with* $\pi_1 \leq \pi_2$ *(*error-minimality*) and* $\theta(\phi_2) \equiv_{\pi_1} \phi_1$ *(*principality*),* $\theta(\mathcal{A}_2) \equiv_{\pi_1} \mathcal{A}_1$, *and* $\theta(\Delta_2) \equiv_{\pi_1} \Delta_1$ *for some* θ. *A similar result holds for* $\vdash_M$ *and inferM.*

In the theorem, we extend the relation $\equiv_\pi$ to assumption sets and choice environment by applying it to their corresponding type components. This theorem can be proved with an induction over the typing rules in Fig. 6.

7. Coarse incremental type inference

The type of *infer1*, $e \to \pi \times \mathcal{A} \times \phi \times \Delta$, indicates that the expression e decides the type and the assumption set of e. This immediately shows that if a subexpression hasn't been changed, then there is no need to perform type inference for that subexpression. In particular, when a node is changed, the type information for only the path from that node to the root needs to be recomputed. For example, if we change `True` in `rank` to `1`, we need to update the type information for the path from the right-most node to the root. This allows us to recompute type information for four nodes only, rather than eight nodes if we recompute everything from scratch.

With some tricks, we can save more computations for updating error fixes. The first trick is to reuse fresh variables generated for nodes. Consider, for example, now we want to change the first occurrence of x in `rank` to y. Note that in Fig. 8 we generated fresh type variables α_7 and α_1 and the fresh choice name A for `x`. If we generate other fresh names for y, the left application node will detect that the type of the function has been changed, the type of the application thus has to be recomputed. This recomputation, however, can be avoided by reusing the fresh names at y. Fig. 9 depicts this process, where we reuse fresh names α_7, α_1, and A for y. As a result, we can reuse the type for the left @ without solving the unification problem U_1. This allows us to further reuse the result type of the node $(,)$.

The second trick is reordering the process of unifying all assumptions for the same variable, which is done at (1c) in *infer1*. Given n types $\phi_1, \phi_2, \phi_3, \ldots, \phi_n$, *infer1* goes through $n-1$ iterations to unify them. It first unifies ϕ_1 and ϕ_2, yielding a unifier η_1 and a typing pattern π_1. Next, the types ϕ_2 and ϕ_3 are substituted with η_1 and then unified, producing another unifier and typing pattern to be used in the next iteration. This continues until all types are unified. It's easy to see that later iterations depend on previous iterations, meaning that if an assumption is changed, then all the unifications in later iterations are invalidated.

We address this problem by first unifying the first assumption with each of the rest assumptions, yielding $n-1$ unifiers and $n-1$ typing patterns. These unifiers are combined into a new unifier through *substitution combination*, a technique developed in [31, §6]. Given a set of substitutions $S = \{\eta_1, \eta_2, \ldots, \eta_n\}$, substitution combination $C(S)$ returns a substitution η and a typing pattern π such that

$$\forall \phi_1 \phi_2 : \forall \eta_i \in S : \eta_i(\phi_1) \equiv_\pi \eta_i(\phi_2) \Rightarrow \eta(\phi_1) \equiv_\pi \eta(\phi_2)$$

In this method, an assumption change will invalidate $n-1$ unifications if the first assumption is changed and only 1 unification if some other assumption is changed.

We refer to the inference algorithm implementing the ideas in this section *infer2*. We will omit its presentation since it is very similar to *infer3* in Section 8.2. In particular, *infer2* can be obtained from Fig. 11 by replacing all occurrences of *infer3* with *infer2* and replacing function calls of the form $\mathit{incrVU}(\phi_1 \equiv^? \phi_2, U', (\pi', \theta'))$ with calls of the form $\mathit{vunify}(\phi_1, \phi_2)$.

8. Refined incremental type inference

Incremental type inference in Section 7 tries to maximize the reuse of typing information, thus minimizes the number of variational unification problems to be resolved. However, once a type is changed, the unification problems involving that type have to be resolved, even the change is minor. For example, consider again changing `True` to `1` in `rank`, which causes the leaf to have the type `Int`. Based on previous subsection, we have to update the type information for four nodes, which needs to solve the following unification problems U_2' and U_3'.

$$U_2' : D\langle \alpha_8, \alpha_4 \rangle \equiv^? E\langle \texttt{Int}, \alpha_5 \rangle \to \beta_2$$
$$U_3' : A\langle B\langle \texttt{Char}, \alpha_2 \rangle \to \beta_1, \kappa_1 \rangle \equiv^? D\langle E\langle \texttt{Int}, \alpha_5 \rangle \to \beta_2, \kappa_3 \rangle$$

We observe that U_2' is very similar to U_2: the only difference is that `Bool` in the right-hand-side (RHS) of U_2 is changed to `Int` in U_2'. The change from U_3 to U_3' is the same. Since type unification is a main part of type inference and needs intensive computations, a natural question is, can we reuse unification results to solve unification problems incrementally? The answer is yes. We present in Section 8.1 such a method and in Section 8.2 a refined type inference algorithm using this method.

8.1. Incremental variational unification

The general idea of incremental variational unification is that we first compute the difference between two unification problems, yielding a delta unification problem, which is then solved and the result is merged into the original result to get a unifier for the new unification problem.

Given two unification problems $U : \phi_l \equiv^? \phi_r$ and $U' : \phi_l' \equiv^? \phi_r'$ and the result (π, θ) for U, we take the following steps to solve U'.

Compute differences To compute the difference between two unification problems, we first need to compute that between two types. We use the function $\mathcal{D}(\phi_1, \phi_2)$ to compute the difference between ϕ_1 and ϕ_2. The result is a set of decisions, where each decision δ satisfying $\lfloor \phi_1 \rfloor_\delta \not\equiv \lfloor \phi_2 \rfloor_\delta$. Here are several examples of applying $\mathcal{D}$.

$$\mathcal{D}(A\langle \texttt{Int}, \texttt{Char} \rangle, \texttt{Char}) = \{\{A.1\}\}$$
$$\mathcal{D}(E\langle \texttt{Bool}, \alpha_5 \rangle \to \beta_2, E\langle \texttt{Int}, \alpha_5 \rangle \to \beta_2) = \{\{E.1\}\}$$
$$\mathcal{D}(\texttt{Bool}, \texttt{Bool}) = \{\}$$
$$\mathcal{D}(\texttt{Int}, \texttt{Bool}) = \{\{\}\}$$

The result of the first example indicates that two arguments differ in only one decision, the first alternative of A. The first argument has `Int` while the second argument has `Char` in that decision. The second example is similar although slightly complicated. When two types are the same, the result is an empty set, meaning that there is no decision such that selecting them with the decision results in different types. The third example falls in this case. When the two types are completely different, the result set has one member, which is itself an empty set. Remember that selecting a type with an empty decision gives that type back.

The function $\mathcal{D}(\phi_1, \phi_2)$ is defined as follows.

$$\mathcal{D}(\tau, \tau) = \{\}$$
$$\mathcal{D}(\tau_1, \tau_2) = \{\{\}\}$$
$$\mathcal{D}(\phi_1 \to \phi_2, \phi_3 \to \phi_4) = \mathcal{D}(\phi_1, \phi_3) \cup \mathcal{D}(\phi_2, \phi_4)$$
$$\mathcal{D}(d\langle \phi_1, \phi_2 \rangle, d\langle \phi_3, \phi_4 \rangle) = \{A.1 : \delta \mid \delta \in \mathcal{D}(\phi_1, \phi_3)\} \cup \{A.2 : \delta \mid \delta \in \mathcal{D}(\phi_2, \phi_4)\}$$
$$\mathcal{D}(d\langle \phi_1, \phi_2 \rangle, \phi) = \mathcal{D}(d\langle \phi_1, \phi_2 \rangle, d\langle \lfloor \phi \rfloor_{d.1}, \lfloor \phi \rfloor_{d.1} \rangle)$$
$$\mathcal{D}(\phi, d\langle \phi_1, \phi_2 \rangle) = \mathcal{D}(d\langle \phi_1, \phi_2 \rangle, \phi)$$

The first two cases are straightforward. The third case computes the difference between two function types by considering their respective argument and return types. To compute the difference of two choice types with the same choice name d, $\mathcal{D}$ first computes the difference of the first alternatives of the choices and then adds $d.1$ to each decision. Similar, $\mathcal{D}$ does this for the right alternatives and extends the results with $d.2$. In the remaining cases, one argument is a choice type and the other is not. These cases are reduced to the fourth case.

We overload $\mathcal{D}$ to compute difference between two unification problems and define $\mathcal{D}(U, U')$ as $\mathcal{D}(\phi_l, \phi_l') \cup \mathcal{D}(\phi_r, \phi_r')$. With this definition, we have $\mathcal{D}(U_2', U_2) = \{\{E.1\}\}$ and $\mathcal{D}(U_3', U_3) = \{\{D.1, E.1\}\}$.

Resolve subproblems Each decision δ in $\mathcal{D}(U, U')$ represents a unification problem $\lfloor U' \rfloor_\delta$ to be resolved. We can again use *vunify* to solve these problems. However, to simplify the operation of the next step, we use *vunify*$'$, a variant of *vunify*. The main difference lies in the way they represent unification results. Given a unification problem $\phi_1 \equiv^? \phi_2$, *vunify* returns (π, θ) while *vunify*$'$ returns $\{(\delta, \pi, \theta)\}$, a set of triples. Each triple (δ, π, θ) satisfies that (1) *vunify*$(\lfloor \phi_1 \rfloor_\delta, \lfloor \phi_2 \rfloor_\delta) = (\pi, \theta)$ and (2) *vunify*$(\lfloor \phi_1 \rfloor_\delta, \lfloor \phi_2 \rfloor_\delta)$ doesn't need to decompose any choice. Consider, for example, the unification problem $A\langle \texttt{Int}, \alpha \rangle \equiv^? \texttt{Int}$. While *vunify* returns $(\top, \{\alpha \mapsto A\langle \kappa, \texttt{Int} \rangle\})$, *vunify*$'$ returns $\{(\{A.2\}, \top, \{\alpha \mapsto \texttt{Int}\})\}$. As another example, consider U_2 from Fig. 8. For U_2, *vunify* returns $(\top, \theta_2)$, where

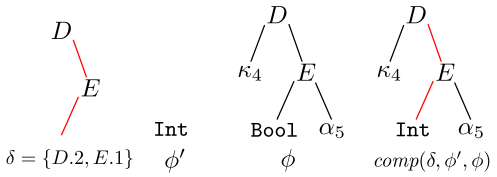

Fig. 10. An example of *comp*.

$$\theta_2 = \{\alpha_8 \mapsto D\langle E\langle \texttt{Bool}, \alpha_5\rangle \to \beta_2, \kappa_3\rangle, \alpha_4 \mapsto D\langle \kappa_4, E\langle \texttt{Bool}, \alpha_5\rangle \to \beta_2\rangle\}$$

while *vunify*′ returns

$$\{(\{D.1\}, \top, \{\alpha_8 \mapsto E\langle \texttt{Bool}, \alpha_5\rangle \to \beta_2\}),$$
$$(\{D.2\}, \top, \{\alpha_4 \mapsto E\langle \texttt{Bool}, \alpha_5\rangle \to \beta_2\})\}$$

The definition of *vunify*′ differs from *vunify* in how it returns results only, and we will not present the definition in detail. Based on *vunify*′, we define $\mathcal{R}(U, U')$ to solve all the unification subproblems and collect all the results as follows,

$$\mathcal{R}(U', U) = \bigcup_{\delta \in \mathcal{D}(U,U')} \{(\delta \cup \delta', \pi, \theta) \mid (\delta', \pi, \theta) \in \mathit{vunify}'(\lfloor \phi'_l \rfloor_\delta, \lfloor \phi'_r \rfloor_\delta)\}$$

As an example, the result of $\mathcal{R}(U'_2, U_2)$ is

$$\{(\{D.1, E.1\}, \top, \{\alpha_8 \mapsto \texttt{Int} \to \beta_2\})(\{D.2, E.1\}, \top, \{\alpha_4 \mapsto \texttt{Int} \to \beta_2\})\}$$

Merge results Given each $\{(\delta', \pi', \theta')\}$ from $\mathcal{R}(U, U')$ and (π, θ) for U, we can merge them together to get the result for solving U'. An intuitive way to understand merging is view each variational type (pattern) as a tree and the merging process is replacing the subtrees of the variational type (pattern) with given subtrees. The merging process is adapted from the function *comp* from [14, §7.2].

Given δ, ϕ', ϕ, $\mathit{comp}(\delta, \phi', \phi)$ replaces the type at δ in ϕ with ϕ' and leaves other parts unchanged. We can view ϕ as a tree with the root on the top and δ as a path from the root to an internal node or a leaf, and $\mathit{comp}(\delta, \phi', \phi)$ replaces the subtree specified by δ with ϕ'. For example, Fig. 10 gives an example of applying $\mathit{comp}(\delta, \phi', \phi)$ with $\delta = \{D.2, E.1\}$ and $\phi = D\langle \kappa_4, E\langle \texttt{Bool}, \alpha_5\rangle\rangle$. For ϕ, the tree representation has internal nodes D and E and has three leaves. For the path specification δ, as it includes $D.2$, it means that the path takes the second children of D. As it includes $E.1$, it means that the path takes the first children of E. Overall, the result of *comp* replaces `Bool`, the subtree at $\{D.2, E.1\}$, with `Int`.

The function *comp* is defined as follows. When it is called, the cases are tried top-down and the first case that matches will be executed. In the first case, we replace ϕ with ϕ, and we immediately return ϕ because the replacing does not make any change. In the second case, if both types are function types, then *comp* is recursively applied to the corresponding parameter types and return types.

$$\begin{aligned}
\mathit{comp}\,(\delta, \phi, \phi) &= \phi \\
\mathit{comp}\,(\delta, \phi_1 \to \phi_2, \phi_3 \to \phi_4) &= \mathit{comp}\,(\delta, \phi_1, \phi_3) \to \mathit{comp}\,(\delta, \phi_2, \phi_4) \\
\mathit{comp}\,(\{d.1\}, \phi', d\langle \phi_1, \phi_2\rangle) &= d\langle \phi', \phi_2\rangle \\
\mathit{comp}\,(\{d.2\}, \phi', d\langle \phi_1, \phi_2\rangle) &= d\langle \phi_1, \phi'\rangle \\
\mathit{comp}\,(\{d.i\}, \phi', \phi) &= \mathit{comp}\,(\{d.i\}, \phi', d\langle \phi, \phi\rangle) \\
\mathit{comp}\,(d.1 : \delta, \phi', d\langle \phi_1, \phi_2\rangle) &= d\langle \mathit{comp}\,(\delta, \phi', \phi_1), \phi_2\rangle \\
\mathit{comp}\,(d.2 : \delta, \phi', d\langle \phi_1, \phi_2\rangle) &= d\langle \phi_1, \mathit{comp}\,(\delta, \phi', \phi_2)\rangle \\
\mathit{comp}\,(d.i : \delta, \phi', \phi) &= \mathit{comp}\,(d.i : \delta, \phi', d\langle \phi, \phi\rangle)
\end{aligned}$$

Cases three through five handle the situation that the δ contains only one selector. The selector decides which alternative in ϕ will be replaced with ϕ'. The last three cases handle the situation that the decision has more than a selector. We use the syntax $d.i : \delta$ to single out a selector $d.i$ and bind the rest of the decision to δ. The selector $d.i$ decides the child in which the *comp* will be recursively applied.

Given each $\{(\delta', \pi', \theta')\}$ from $\mathcal{R}(U, U')$ and (π, θ) for U, we can merge them together to get the result for solving U' as follows. First, to merge π' into π with the decision δ', we call $\mathit{comp}\,(\delta', \pi', \pi)$. We can merge θ' into θ with δ' similarly. For each $\alpha' \mapsto \phi'$ in θ', if $\alpha' \in \mathit{dom}(\theta)$, we merge ϕ' into $\theta(\alpha')$ with δ' through $\mathit{comp}\,(\delta', \phi', \theta(\alpha'))$. If $\alpha' \notin \mathit{dom}(\theta)$, we simply add $\alpha' \mapsto \mathit{comp}(\delta', \phi', \kappa)$ to θ, where κ is a fresh type variable.

Now we can merge $\mathcal{R}(U_2, U'_2)$ into θ_2 (from previous step), yielding $(\top, \theta'_2)$, where

$$\theta'_2 = \{\alpha_8 \mapsto D\langle E\langle \texttt{Int}, \alpha_5\rangle \to \beta_2, \kappa_3\rangle, \alpha_4 \mapsto D\langle \kappa_4, E\langle \texttt{Int}, \alpha_5\rangle \to \beta_2\rangle\}$$

```
infer3 : e → π × 𝒜 × φ × Δ

(3a) infer3(x) =
    φ ← d°⟨α₁°, α₂°⟩                                  {- reuse fresh names -}
    return (⊤, {(x, α₁°, d°)}, φ, {(ℓ(x), φ)})

(3b) infer3(e₁ e₂) =
    (π₁, 𝒜₁, φ₁, Δ₁) ← infer3(e₁)
    (π₂, 𝒜₂, φ₂, Δ₂) ← infer3(e₂)
    if φ₁° = φ₁ and φ₂° = φ₂
        return (π° ⊗ π₁ ⊗ π₂, θ°(𝒜₁ ∪ 𝒜₂), φᵣ°, θ°(Δ₁ ∪ Δ₂))
    (π, θ) ← incrVU(φ₁ ≡? φ₂ → β°, U°, (π°, θ°))
    return (π ⊗ π₁ ⊗ π₂, θ(𝒜₁ ∪ 𝒜₂), θ(β), θ(Δ₁ ∪ Δ₂))

(3c) infer3(λx.e) =
    (π, 𝒜, φ, Δ) ← infer3(e)
    {φ₁, φ₂, ..., φₙ} ← 𝒜ᵀ(x)
    for each φᵢ in {φ₂, ..., φₙ}
        (πᵢ, θᵢ) ← incrVU(φ₁ ≡? φᵢ, Uᵢ°, (πᵢ°, θᵢ°))
    (πᵤ, θ) ← C({θ₂, ..., θₙ})
    πₐ = ⊗_{i∈{2,...,n}} πᵢ
    return (π ⊗ πᵤ ⊗ πₐ, θ(𝒜 \ x), θ(φ₁ → φ), θ(Δ))
```

Fig. 11. A refined incremental inference algorithm that uses incremental variational unification.

Incremental variational unification Based on the three steps discussed above, we can define the function *incrVU* to incrementally solve U' based on the result (π, θ) for U, where $\mathcal{M}$ denotes the merging process described in the third step.

$$incrVU(U', U, (\pi, \theta)) = \mathcal{M}(\mathcal{R}(U', U), (\pi, \theta))$$

Instantiating U' to U_2', U to U_2, π to $\top$, and θ to θ_2 in *incrVU*, we obtain θ_2' as the solution for U_2'. If we solve U_2' with *vunify*, then we obtain the same result as U_2'. In general, we have the following correctness result, where $vunify(U)$ solves U with the variational unification algorithm [14].

Theorem 4 *(incrVU correctness). Given U and U', let $(\pi, \theta) = vunify(U)$, then $vunify(U') = incrVU(U', U, (\pi, \theta))$.*

The proof of this theorem is given in the companion [11] of this paper.

8.2. A refined incremental inference algorithm

Based on *incrVU*, we present *infer3*, a refined incremental type inference algorithm, in Fig. 11. In the figure, we don't present the case for **let** expressions since it can be translated from case (1d) in a similar way as we did for (3c) (Fig. 11) from (1c) (Fig. 7). In the figure, we use the notation o° to denote the saved copy of o from the last run of *infer3*. If there is no saved value, then a meaningful value is returned. For example, in case (3a), we use d° to return the choice name generated last time. However, if no fresh choice name was generated and saved, then d° just generates a new fresh choice name and returns it. If U° doesn't exist, then the corresponding call of *incrVU* will call *vunify* instead.

In the figure, we omitted the logic of using a dirty flag to denote if an expression is changed and propagate the flag to its parent. If an expression is not changed, *infer3* simply propagate this information to its parent and will not perform any computations.

We now briefly go through each case. Case (3a) is very similar to (1a). The only difference is that we try to reuse fresh names as much as possible. Case (3b) types applications. It first checks if both the function type and the argument type are the same as saved copies. If so, then the saved π° and θ° are used without unifying the function type and the argument type. Otherwise, it uses the incremental variational unification algorithm to solve the new unification problem. Finally, case (3c) deals with abstractions. It uses the idea of substitution combination introduced in Section 7 to reorder the process of unifying all assumptions. This helps to reuse unification results.

9. Evaluation

To test the feasibility of *e*CFT, we have developed a prototype that implements the ideas from this paper in Haskell. The prototype supports all three inference algorithms: *inferM* (Section 6), *inferM* calling *infer2* (Section 7) instead of *infer1*, and *inferM* calling *infer3* (Section 8.2), which we will refer to as *recomputing*, *coarse*, and *refined*, respectively. In addition to the constructors presented in Fig. 5, our prototype also supports data types and case expressions. Although the formalizations in the paper don't consider predefined global variables, the prototype supports them by resolving unbound variables left in $\mathcal{A}$ in the predefined type environment and introducing errors when the resolution fails.

The prototype reuses two components from CFT: heuristics for ranking error suggestions and the algorithm for deducing expression changes from type changes.

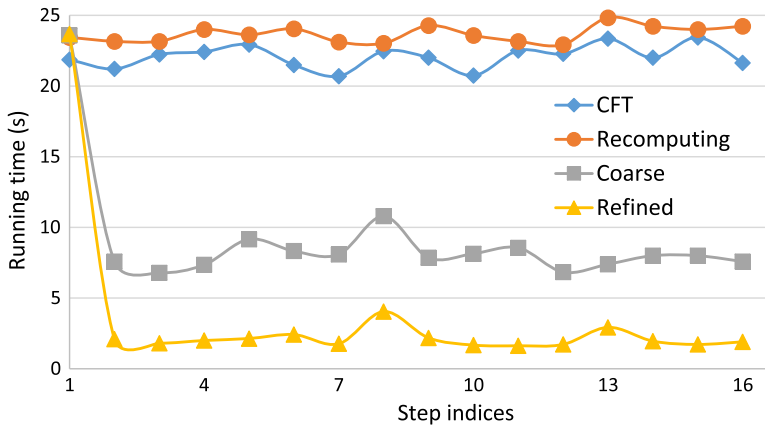

Fig. 12. Running time of CFT and different inference algorithm of *e*CFT on student programs.

Section 5 shows that *e*CFT and CFT produces the same set of error fixes for any given expression. We have experimentally confirmed this by running both *e*CFT and CFT prototypes over the benchmark we collected before [7]. The benchmark contains 121 programs from 22 publications in the literature. For this reason, our evaluation focuses on performance.

9.1. Running time against program steps

Our first performance test used 60 program sequences from the student program databases [23,50]. The initial sizes of these sequences range from 47 to 136 LOC and the numbers of steps range from 5 to 42. The results for different sequences are quite similar and we present the result for a representative sequence (whose initial LOC is 125 and the number of updates is 15) in more detail. The ratio of the difference between two consecutive program versions over the original version ranges from 0.01 to 0.1 (except for the step 8 where the ratio is 0.14). Fig. 12 presents the running time of CFT and three inference algorithms of *e*CFT. The times are measured on a laptop with a processor having four 2.4 GHz dual-cores and 8 GB RAM running 64-bit Ubuntu 16.04 LTS and GHC 8.0.2.

The response time (the time delay to display the first error message) of CFT is about 22.3 s in average for this sequence. *e*CFT is up to 3× faster with *coarse* and 13× faster with *refined*. With *refined*, the response time ranges from 1.7 s to 2.4 s except for the step 8, where the response time is 4.0 s. The reason is that the change ratio is 0.14 at that step, much higher than those at other steps. The fact that *coarse* is much slower than *refined* reflects that unification problems are getting more and more complex as type inference moving up in the AST, although the change in each unification problem may be minor. This demonstrates the value of *refined*.

Fig. 12 shows that *recomputing* is in average 1.4 s slower than CFT. There are two potential reasons for this. First, for unbound variables, CFT directly detects the problem at variable references whereas *recomputing* first assumes that variables are always bound and have to adjust the typing result after the typing is completed. Second, for variables predefined globally, such as list processing operations, CFT immediately gets the types at variable references but *e*CFT first assigns fresh type variables to variables and record them in $\mathcal{A}$. In general, $\mathcal{A}$ in *e*CFT contains much more items than Γ in CFT so looking up $\mathcal{A}$ is more time consuming.

At step 1, *coarse* and *refined* can't reuse any previous error fixes since they don't exist yet. As a result, they take slightly more time than *recomputing* since they need to test if results have already computed. This can be seen from Fig. 12.

For other tested sequences, the speedup of *refined* over CFT ranges from 4.2× to 19.1× and is more than 12.4× in 80% cases. For *coarse*, the speedup over CFT ranges from 1.2× to 5.3× and is more than 2.6× in 80% cases.

The previous test investigated how *e*CFT performs on real sequences of program updates, and such programs are relatively small. We are also interested in investigating how *e*CFT performs in larger programs. Ideally, we could use some existing benchmark for such an investigation, such as git commits and large applications. However, a main challenge to this idea is that usually publicly available applications do not contain type errors since they are well typed. Also, the differences between consecutive commits are often bigger than those programmers make during interactive program debugging and the former thus may not reflect real-world error debugging situations well. Finally, real world applications often use language features that are not provided by the prototype, such as type classes, type families, etc. [48].

For these reasons, we instead synthesized large programs by combining programs from different program sequences. Specifically, in our experiment, we took three program sequences from the student program database, and created a new sequence where each program is a combination of corresponding programs from these three sequences (necessary renaming was done to avoid name conflicts). The programs in this sequence have a trend of getting bigger because in addition to debug type errors, some new functions were added by students. The LOCs of these programs range from 331 to 409.

We present the running times for this sequence in Fig. 13. In general, we can observe that the running times for CFT and *recomputing* are more sensitive to program size increase than *coarse* and *refined* do. For example, as the program size

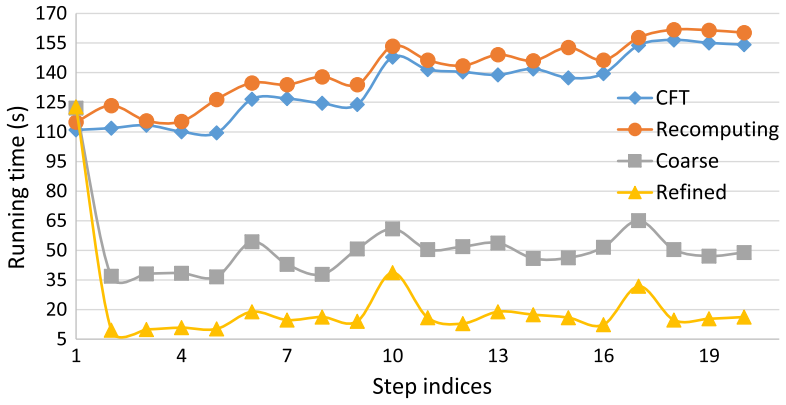

Fig. 13. Running time of CFT and different inference algorithm of *e*CFT on large, synthesized programs.

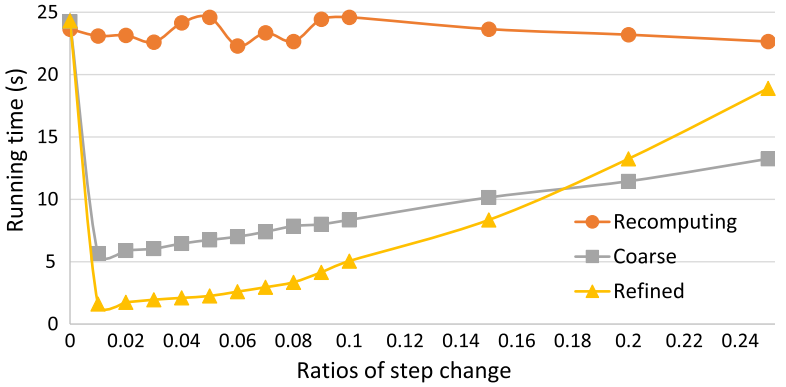

Fig. 14. Running time of different inference algorithm of *e*CFT on synthesized programs.

experienced three quite significant increases, the running times of CFT and *recomputing* follow a similar pattern. In contrast, the running times of *coarse* and *refined* increase quite significantly just once and then become quite stable. Also, we observe that *refined* is much faster than CFT. It shortens the response time by about 89% for all steps except for those where program sizes increased significantly, when no previous fixes for new code could be reused.

9.2. Running time against change ratios

In practice, programs may be updated very differently for debugging type errors. For this reason, we have conducted another test, where we chose another 60 student programs of 71 to 142 LOC [23]. For each of the change ratios from 0.01 to 0.1 with an interval of 0.01 and from 0.1 to 0.3 with an interval of 0.05, we randomly generated changes to each original program. (We used the Haskell package `haskell-src` to first parse the source code to an AST, make changes to the AST, and then generate the source code from the AST.) Changes led to ill-formed programs were immediately discarded. We stopped until we had generated 50 unique changes that all yield well-formed programs for each ratio for each program.

Since the results for the tested programs exhibit a very similar pattern, we report the result for the original program that has 93 LOC in Fig. 14 and omit the details for other programs. Each time in the figure is an average of the randomly generated 50 programs at that ratio.

The result is similar to that in Fig. 12 when the change ratio is less than 0.1. The result is somewhat surprising when the ratio is high. As the change ratio increases, the running time for *coarse* increases almost linearly. However, the time increases much faster to the increase of change ratio for *refined*.

While we don't have the exact reason why this happens, we give a possible explanation here. According to the definition of $\mathcal{D}$, the incremental variational unification doesn't take the advantages of variational typing. For each decision δ in $\mathcal{D}(U, U')$, a subproblem $\lfloor U' \rfloor_\delta$ is solved. The subproblems may overlap, yielding many smaller subproblems to be solved repeatedly. This gets worse as the change ratio becomes larger. In particular, when the change ratio is 1, the difference covers all decisions and *refined* degrades to the brute-force approach of finding all error fixes without using variational typing to reuse typing information.

Why this sub-exponential growth doesn't show up while the change ratio is low? There are two possible reasons. While the ratio is low, the computations that are wasted are very limited. Also, at each point, the time for ranking all error fixes

and for deducing expression changes is similar. When the ratio is low, the time for this part is more significant than the time for update error fixes. As a result, this growth pattern is maybe shaded.

Ideally, *refined* is always (much) faster than *coarse*, but this is not the case as this test demonstrates. We are interested in combining them into another algorithm that is always as fast as the better algorithm for different change ratios. One parameter we can use is the change ratio. We, however, leave this for future work.

Nevertheless, the result of *refined* is very promising when we considering both Figs. A.16 (Appendix A) and 14. Fig. A.16 shows that for more than 80% of changes, the change ratio is below 0.1, and Fig. 14 shows that *refined* is still very fast when the change ratio reaches 0.1. The probability of high change ratio is very low, which means that *refined* has low performance in very few cases.

10. Related work

Type error debugging has been extensively studied [51,29,49,7,57,39,4]. In [7], we have discussed the relation of CFT with much previous work, such as discriminative sum types [36], Seminal [30], and Chameleon [46]. These discussions also apply here and thus in this paper we mainly discuss the relation with other work. We group our discussions based on the properties these approaches share.

10.1. Principal typing for debugging type errors

Our type system is based on principal typing, which is very different from principal types [52]. There have been many other error debugging approaches based on principal typing.

Chitil [16] suggested that the main difficulty of understanding why a program is ill typed lies in knowing why subexpressions get certain types. He further argued that the core of the difficulty in HM is that the type system is not compositional. Based on this observation, he developed a compositional error explanation approach using principal typing. His approach allows the user to navigate through the explanation graph and inspects the type of each node. By asking the user whether the type of each node is intended, his approach can potentially improve the precision of error localization. While our approach focuses on each potential erroneous expression once a time and provides a detailed error message, his approach doesn't provide change suggestions but allow the user to have a big-picture about why errors have occurred. In this sense, these approaches are complementary to each other.

Helium [27] is a compiler designed for Haskell beginners. It aims to provide good quality type error messages. Helium uses a constraint-based type inference algorithm to generate a set of type constraints, and then uses a solver to handle all the constraints globally. When the constraints are unsatisfiable, it uses a set of heuristics to find the most suspicious constraint [27,24], from which a few most likely error sources are identified. In this approach, it is important to keep the constraints from different parts of the expression independent. For this reason, the order it traverses the AST is similar to what principal typing does.

Haack and Wells [22] computed program slices that identify all program locations contributing to type errors. Their approach first uses Damas's type inference algorithm T [17] to generate type constraints, and then found the minimal unsolvable subsets from the type constraints, from which program locations consisting of the error slice are computed. The underlying type system T used in the approach can be viewed as a variant of principal typing [28].

While these approaches employed principal typing to make typing compositional or independent among different expression parts, we use principal typing to enable us to incrementally update error fixes. Another main difference is that our approach provides a detailed change suggestions for all possible error causes while other approaches don't do this or do so for a few locations.

10.2. Other error debugging approaches

Reordering unifications and heuristics There have been many approaches to improve the precision of error localizations. Algorithm $\mathcal{W}$ is biased in the order of solving unification problems. Consequently the accuracy of error localizations is affected. Many approaches [18,34] altered the ordering of type unifications to eliminate the bias in $\mathcal{W}$. All these proposed algorithms followed the idea of $\mathcal{W}$ in the sense that they treated the place where unification failed as the error cause. Unfortunately, the reported error location may be far from the real error cause in algorithms using a fixed ordering for unifying types.

Some approaches [29,26,24] used heuristics to select the most likely error cause from a list of candidate locations. Heuristics work well when the context information is abundant and not so well when little such information is available [7].

Type error slicing Another extreme to these approaches is the idea of finding all error locations that may contribute to the type error [22,42]. Error slicing approaches never miss the real error cause, but their value diminishes when they cover too many locations in the program, relying on the user to pinpoint the proper error location [27]. In addition, error slicing approaches do not provide detailed error messages. Our type inference algorithm computes all potential type changes to remove type errors. Comparing with slicing approaches, the changes we present cover fewer locations. Users can look at one location and the corresponding suggestions for type changes.

Type error explanations Many approaches have developed to generate explanations about why type errors have occurred [51,55]. They were realized by extending type inference algorithms to record why certain types are unified with each other. These approaches are very different to our approach in both realization techniques and behaviors.

In general, ill-typed programs can not be run. Seidel et al. [43], however, have developed a special language interpreter that is able to evaluate ill-typed programs. Given a program, the interpreter could show intermediate evaluation results and illustrates to the programmers where evaluation gets stuck due to type errors. Such information is usually more tangible to programmers and helps them understand where the real error causes are. That idea, however, usually shows just one error location (where evaluation is stuck) while CFT and *e*CFT can find all error locations.

Finding most likely error causes Given an ill-typed expression, SHErrLoc [56,57] first extracts typing constraints (such as different occurrences of a single parameter should have the same type, the parameter type must match the type of the argument in a function application, the condition must have the type `Bool` and the branch types must be the same of a conditional), translates the typing constraints into a graph representation, and localizes the most likely error causes using a simple Bayesian model. The basic idea is that in the graph representation, each path is classified as consistent or inconsistent, and a node (which corresponds to some subexpression in the source program) in the graph is identified as an error cause if it touches as few as consistent paths and as many as inconsistent paths. This idea coincides with the belief that, when debugging type errors, the changes to the program should be as few as possible. The main difference between SHErrLoc and CFT and *e*CFT is that SHErrLoc usually reports type errors at constraint level, which could be hard to make use of by programmers, while CFT generates more concrete error messages, at the type level and sometimes even at the expression level. Another difference is that while SHErrLoc finds a few most likely error causes, CFT and *e*CFT could find all of them.

Instead of developing special graph representations to represent typing constraints, MinErrLoc [38,39] expresses typing constraints as SMT formulas [37]. Essentially, MinErrLoc reduces the type inference problem to SMT constraint solving problem. If SMT solving for a program succeeds, then type inference for the program succeeds with no type errors. However, if SMT solving fails, then the program contains type errors. In this case, MinErrLoc uses MaxSMT to identify the maximal satisfiable set of SMT constraints and extract the rest of constraints as causing type errors. Similar to CFT and *e*CFT, MinErrLoc can find comprehensive error causes but it does not provide concrete change suggestions.

Instead of using an off-the-self SMT solver, Mycroft [32] explores the constraint solver developed by individual languages to find the most likely error cause. The basic idea is that when solving the full constraint set of a program fails, it partitions that into two subsets and recursively solves them.

Seidel et al. [44] developed a machine learning based approach, named Nate, to locate likely error causes. Nate is trained from a set of more than 6,000 ill-typed student programs with the corresponding terms that are changed to fix the type errors. Given each new program, Nate will predict a few most likely terms that need to be changed based on the trained model. Compared to our approach, Nate's error messages include only location information while CFT and *e*CFT includes more information, including the type the located term should have to remove the type error.

Based on our machinery from variational typing [14] and error-tolerant typing [12], Eremondi et al. [20] have developed a framework for diagnosing type errors for dependently-typed programming languages.

Our earlier work on type error debugging We have explored many different directions for improving type error debugging in addition to CFT [7,10], which has been discussed in detail in Section 1. On top of CFT, we developed a method named Guided Type Debugging (GTD). A main difference of GTD with existing error debugging approaches, including CFT, is that it takes users' intended result types into consideration while others do not. GTD allows users to specify their expected type of the ill-typed expression. It then computes all erroneous locations and their expected types, ranks them, and presents the suggestions to users in that order. As GTD is based on CFT, it suffers from the same issue as CFT does. As a result, we expect that our approach developed in this paper could help solve the problem in GTD as well.

Prior to CFT, we explored the idea of lazy typing [8]. The main idea is that when branches (in **if** and **case** expressions) have inconsistent types, we create variational types whose alternatives are branch types and then continue the typing process, rather than terminating the typing process and choose some branch as having type errors. This idea avoids a premature, uninformed decision and gather information about the context to decide which branch has the type error. The main differences between lazy typing and CFT are: (1) CFT generates a complete set of changes while lazy typing tries to identify the most likely change only and (2) CFT points to very specific change location while lazy typing has only branch-level granularity.

We have explored the idea of combining different error debugging tools for debugging type errors [13]. The main insight is that different tools have different strengths and weaknesses, and they may good at debugging type errors caused by certain reasons but not by others. Combining tools with complementary strengths is likely to yield a better tool. Our work used Helium [27] and lazy typing as studying subjects and manually investigated their type error messages on more than 1,000 ill-typed programs. For each error message, we analyzed its precision (the difference between the reported error location and the real error location) and its concreteness (whether it contains change suggestion at expression level, at type level, or none). We observed that there is a connection between the concreteness and precision. We further analyzed that creating a new tool based on Helium and lazy typing using this observation could improve error reporting accuracy by about 20% over Helium and lazy typing.

We have also explored the idea of using machine learning to improve error locating and generate user-friendly error messages [54]. The motivation there was that in practice fixing type errors requires restructuring programs (adding or

removing parentheses, adding or removing square brackets, pulling certain subexpressions out of or pushing them into parentheses, etc.) but existing debugging methods do not work well for such type errors [53]. That work [54] thus has a different focus point from this one.

10.3. Principal typing and incremental computing

Erdweg et al. [19] developed a method named co-contextual typing for deriving incremental type checkers by replacing the top-down context flow with bottom-up typing flow. They explored many different incrementalization schemes, for example, incremental constraint solving and eager substitution, which keeps the substitutions passed from children to parents to be small. Their approach resolves the whole constraint once it is updated while our approach resolves only part of the constraint (unification problems). The reason is that unification problems can be very complicated in our system as it encodes different possibilities to remove the type error. As another difference, it seems that their approach doesn't exploit the full advantages of principal typing for processing **let** expressions [19, §3.2]. They first compute the type of the bound expression, generalize the type for the variable, and then type the body. We take a different approach by allowing the bound expression and the body to be typed independently.

Johnson and Walz [29] developed an approach to locate the most likely error source based on an idea similar to majority voting. While type constraints are incrementally generated as programs are edited in their approach, type error debugging itself is not incremental. Miao and Siek [35] developed a type system to detect type errors earlier in multi-staged programming, such as C++ Templates. As meta evaluations are performed, new typing constraints are generated and added to those generated in earlier stages and constraint satisfiability is checked. While in our approach type constraints maybe changed, in their approach existing constraints are never changed.

Acar et al. [1,15] have developed frameworks for performing general incremental computing. We didn't use their framework and developed our own incremental algorithm for several reasons. On one hand, as reported in [15,19], such frameworks usually impose high overhead. The reason is that they record detailed computing process and not only the results. On the other hand, they don't support domain-specific optimizations. For example, through reusing fresh type variables, a change from $\lambda x.x$ to $\lambda y.y$ will not change the type of the expression in our approach (both of them have the type $\beta \to \beta$), which is hard to achieve in their frameworks.

11. Conclusions

We have presented *e*CFT, a method to improve the performance of our previous type error debugging method CFT. While CFT is quite effective in locating type errors and providing change suggestions, it has a long response time. To address this problem, we redesigned the type system and used principal typing to compute all error fixes. We have also developed two incremental methods for efficiently updating error fixes as programs are changed. Our evaluation result shows that in average *e*CFT is 12.4× faster than CFT in more than 80% of cases. The response time drops from about 22.3 s in CFT to about 1.7 s in *e*CFT for programs with about 125 LOC in our evaluation. In the future, we plan to investigate how we can improve the performance of *e*CFT when programs are under significant changes.

CRediT authorship contribution statement

Sheng Chen: Conceptualization, Investigation, Methodology, Software, Supervision, Validation, Writing - review & editing. **Baijun Wu:** Conceptualization, Data curation, Investigation, Software, Validation, Writing - original draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

We thank Jurriaan Hage for sharing his collection of student Haskell programs with us. This work is partially supported by The National Science Foundation under the grant CCF-1750886.

Appendix A. Statistical findings of debugging

This section presents our findings about how students debugged type errors. We studied the program database collected at the Utrecht University [23]. The programs were written by students learning Haskell using the Helium compiler [26]. Whenever a program is compiled, Helium saved a copy of the program with a timestamp [25]. By compiling the program, we can decide if it is well typed. This allows us to figure out how many steps are required to remove the type error and how big is the difference between two consecutive versions. In this work, we use a total of 23529 collected programs from 367 different submissions.

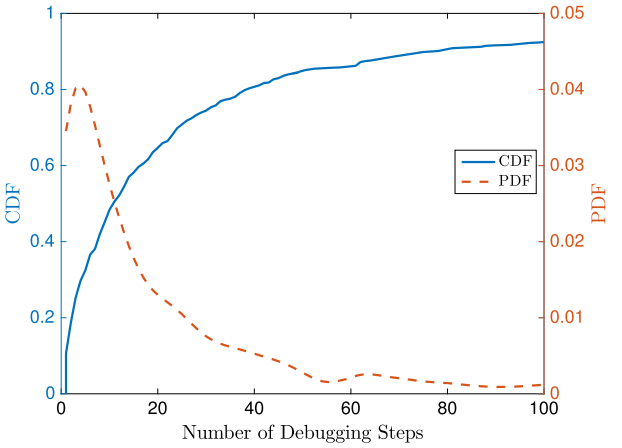

Fig. A.15. Statistics of number of debugging steps.

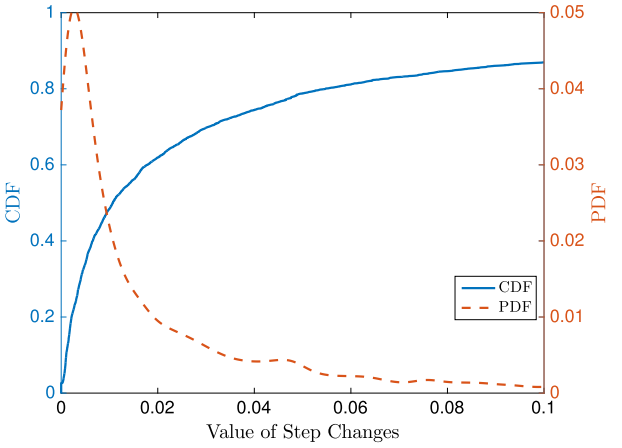

Fig. A.16. Statistics of step changes.

Fig. A.15 presents the statistics about number of steps needed to remove the type error. Given an ill-typed program P_1, if P_0 is well typed and P_n is the first program that is well typed after P_n, then the number of debugging steps for P_1 is $n-1$. We give both the probability density function (PDF) and the cumulative distribution function (CDF) for a given number of debugging steps. For example, a point (x, y) on the PDF curve denotes that the likelihood that the type error is removed using exactly x steps is y. Similarly, (x, y) on the CDF curve means that the likelihood of removing the type error using up to x steps is y. We cut off the x axis at 100 since the probability that the type error needs to be fixed for more than 100 steps is very low, within 0.001. In addition, Fig. A.15 shows that over 92% of programs can be fixed within 100 steps. The maximum debugging step in the collected data is 359, and the average value of debugging steps is 28.94.

We are also interested in measuring the difference between two consecutive versions of the same program, which we will refer to as step change. To compute the step change, we first find all the different expressions between the two programs, and then use Levenshtein distance[4] to measure the difference between those expressions. This gives us a number of bytes that two programs differ. Given a program P, if it differs with the successive version by n bytes, then the step change for P is calculated as $\frac{n}{\text{size of } P}$. Fig. A.16 presents the statistics about step change. For a similar reason as in Fig. A.15, we cut off the x axis at 0.1 since the probability that the step change exceeds 0.1 is very low. We again show both the PDF and the CDF in the figure. From Fig. A.16, we observe that the probability that the step change is more than 10% is less than 0.001, and more than 80% programs are changed less than 10%. The median value of step change is 0.0107, meaning that the program is changed for only about 1%. To sum up, only a very small portion of the program is modified during the debugging process.

Based on Figs. A.15 and A.16, students, and maybe other Haskell beginners, take quite many steps to remove the type error with minor changes in each step. This indicates the feasibility and the value of incremental error fix generation. Our evaluation result in Section 9 will further substantiate the feasibility.

[4] https://en.wikipedia.org/wiki/Levenshtein_distance.

References

[1] U.A. Acar, G.E. Blelloch, M. Blume, R. Harper, K. Tangwongsan, An experimental analysis of self-adjusting computation, ACM Trans. Program. Lang. Syst. 32 (1) (2009) 3:1–3:53.
[2] S. Aditya, R.S. Nikhil, Incremental polymorphism, in: FPCA, 1991, pp. 379–405.
[3] C. Chambers, S. Chen, D. Le, C. Scaffidi, The function, and dysfunction, of information sources in learning functional programming, J. Comput. Sci. Coll. 28 (1) (Oct. 2012) 220–226.
[4] A. Charguéraud, Improving type error messages in OCaml, in: OCaml Users and Developers Workshops, in: Theoretical Computer Science, vol. 198, 2015, pp. 80–97.
[5] S. Chen, Variational Typing and Its Applications, PhD thesis, Oregon State University, 2015.
[6] S. Chen, M. Erwig, Type-based parametric analysis of program families, in: ICFP, 2014, pp. 39–51.
[7] S. Chen, M. Erwig, Counter-factual typing for debugging type errors, in: POPL, 2014, pp. 583–594.
[8] S. Chen, M. Erwig, Better Type-Error Messages Through Lazy Typing, Technical Report, Oregon State University, 2014, http://ir.library.oregonstate.edu/xmlui/handle/1957/58138.
[9] S. Chen, M. Erwig, Principal type inference for gadts, in: POPL, 2016, pp. 416–428.
[10] S. Chen, M. Erwig, Systematic identification and communication of type errors, J. Funct. Program. 28 (2018) e2, https://doi.org/10.1017/S095679681700020X.
[11] S. Chen, B. Wu, Efficient Counter-Factual Type Error Debugging, Tech Report, 2019, Available at https://people.cmix.louisiana.edu/schen//ws/techreport/incrcft.pdf.
[12] S. Chen, M. Erwig, E. Walkingshaw, An error-tolerant type system for variational lambda calculus, in: ICFP, 2012.
[13] S. Chen, M. Erwig, K. Smeltzer, Let's hear both sides: on combining type-error reporting tools, in: IEEE Int. Symp. on Visual Languages and Human-Centric Computing, 2014, pp. 145–152.
[14] S. Chen, M. Erwig, E. Walkingshaw, Extending type inference to variational programs, ACM Trans. Program. Lang. Syst. 36 (1) (2014) 1:1–1:54.
[15] Y. Chen, U.A. Acar, K. Tangwongsan, Functional programming for dynamic and large data with self-adjusting computation, in: ICFP, 2014.
[16] O. Chitil, Compositional explanation of types and algorithmic debugging of type errors, in: ICFP, 2001.
[17] L. Damas, Type Assignment in Programming Languages, PhD thesis, 1985.
[18] H. Eo, O. Lee, K. Yi, Proofs of a set of hybrid let-polymorphic type inference algorithms, New Gener. Comput. 22 (1) (2004) 1–36.
[19] S. Erdweg, O. Bračevac, E. Kuci, M. Krebs, M. Mezini, A co-contextual formulation of type rules and its application to incremental type checking, in: OOPSLA, 2015, pp. 880–897.
[20] J. Eremondi, W. Swierstra, J. Hage, A framework for improving error messages in dependently-typed languages, Open Computer Sci. 9 (1) (2019) 1–32, https://www.degruyter.com/view/journals/comp/9/1/article-p1.xml.
[21] M. Erwig, E. Walkingshaw, The choice calculus: a representation for software variation, ACM Trans. Softw. Eng. Methodol. 21 (1) (2011) 6:1–6:27.
[22] C. Haack, J.B. Wells, Type error slicing in implicitly typed higher-order languages, in: ESOP, 2003, pp. 284–301.
[23] J. Hage, Helium benchmark programs, 2002–2005, Private communication, 2013.
[24] J. Hage, B. Heeren, Heuristics for type error discovery and recovery, in: IFL, in: LNCS, vol. 4449, 2007, pp. 199–216.
[25] J. Hage, P. van Keeken Neon, A library for language usage analysis, in: SLE, in: LNCS, vol. 5452, 2009, pp. 35–53.
[26] B. Heeren, D. Leijen, A. van IJzendoorn, Helium, for learning haskell, in: Haskell, 2003, pp. 62–71.
[27] B.J. Heeren, Top Quality Type Error Messages, PhD thesis, Universiteit Utrecht, The Netherlands, 2005.
[28] T. Jim, What are principal typings and what are they good for?, in: POPL, 1996, pp. 42–53.
[29] G.F. Johnson, J.A. Walz, A maximum-flow approach to anomaly isolation in unification-based incremental type inference, in: POPL, 1986, pp. 44–57.
[30] B. Lerner, M. Flower, D. Grossman, C. Chambers, Searching for type-error messages, in: PLDI, 2007, pp. 425–434.
[31] C.-k. Lin, Practical Type Inference for the GADT Type System, PhD thesis, Portland State University, 2010.
[32] C. Loncaric, S. Chandra, C. Schlesinger, M. Sridharan, A practical framework for type inference error explanation, SIGPLAN Not. (ISSN 0362-1340) 51 (10) (Oct. 2016) 781–799, https://doi.org/10.1145/3022671.2983994.
[33] G. Marceau, K. Fisler, S. Krishnamurthi, Mind your language: on novices' interactions with error messages, in: Onward!, 2011, pp. 3–18.
[34] B.J. McAdam, Repairing type errors in functional programs, PhD thesis, University of Edinburgh, 2002.
[35] W. Miao, J.G. Siek, Incremental type-checking for type-reflective metaprograms, in: GPCE, 2010, pp. 167–176.
[36] M. Neubauer, P. Thiemann, Discriminative sum types locate the source of type errors, in: ICFP, 2003, pp. 15–26.
[37] R. Nieuwenhuis, A. Oliveras, C. Tinelli, Solving SAT and SAT modulo theories: from an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T), J. ACM (ISSN 0004-5411) 53 (6) (Nov. 2006) 937–977, https://doi.org/10.1145/1217856.1217859.
[38] Z. Pavlinovic, T. King, T. Wies, Finding minimum type error sources, in: OOPSLA, 2014, pp. 525–542.
[39] Z. Pavlinovic, T. King, T. Wies, Practical SMT-based type error localization, in: ICFP, 2015, pp. 412–423.
[40] B.C. Pierce, Types and Programming Languages, MIT Press, 2002.
[41] J.A. Robinson, A machine-oriented logic based on the resolution principle, J. ACM 12 (1) (1965) 23–41.
[42] T. Schilling, Constraint-free type error slicing, in: TFP, Springer, 2012, pp. 1–16.
[43] E.L. Seidel, R. Jhala, W. Weimer, Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong), in: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, New York, NY, USA, Association for Computing Machinery, ISBN 9781450342193, 2016, pp. 228–242.
[44] E.L. Seidel, H. Sibghat, K. Chaudhuri, W. Weimer, R. Jhala, Learning to blame: localizing novice type errors with data-driven diagnosis, Proc. ACM Program. Lang. 1 (OOPSLA) (Oct. 2017), https://doi.org/10.1145/3138818.
[45] Z. Shao, A.W. Appel, Smartest recompilation, in: POPL, 1993, pp. 439–450.
[46] P.J. Stuckey, M. Sulzmann, J. Wazny, Improving type error diagnosis, in: Haskell, 2004, pp. 80–91.
[47] F. Tip, T.B. Dinesh, A slicing-based approach for locating type errors, ACM Trans. Softw. Eng. Methodol. 10 (1) (Jan. 2001) 5–55.
[48] V. Tirronen, S. Uusi-mäkelä, V. Isomöttönen, Understanding beginners' mistakes with Haskell, J. Funct. Program. 25 (2015) 1–31.
[49] K. Tsushima, O. Chitil, Enumerating counter-factual type error messages with an existing type checker, in: PPL, PPL 2014, 2014.
[50] P. van Keeken, Analyzing helium programs obtained through logging, Master's thesis, Utrecht University, October 2006.
[51] M. Wand, Finding the source of type errors, in: POPL, 1986, pp. 38–43.
[52] J.B. Wells, The essence of principal typings, in: Automata, Languages and Programming, Springer, 2002, pp. 913–925.
[53] B. Wu, S. Chen, How type errors were fixed and what students did?, Proc. ACM Program. Lang. 1 (OOPSLA) (Oct. 2017), https://doi.org/10.1145/3133929.
[54] B. Wu, J.P. Campora III, S. Chen, Learning user friendly type-error messages, Proc. ACM Program. Lang. 1 (OOPSLA) (Oct. 2017), https://doi.org/10.1145/3133930.
[55] J. Yang, Explaining type errors by finding the source of a type conflict, in: TFP, Intellect Books, 2000, pp. 58–66.
[56] D. Zhang, A.C. Myers, Toward general diagnosis of static errors, in: POPL, 2014, pp. 569–581.
[57] D. Zhang, A.C. Myers, D. Vytiniotis, S. Peyton-Jones, Diagnosing type errors with class, in: PLDI, 2015, pp. 12–21.